Mejoras a la demostración interactiva de propiedades Alloy utilizando SAT-Solving

Moscato, Mariano Miguel

2013

UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Mejoras a la demostración interactiva de propiedades Alloy utilizando *SAT-Solving*

Tesis presentada para optar al título de
Doctor de la Universidad de Buenos Aires
en el área de Ciencias de la Computación

Mariano Miguel Moscato

Director: Dr. Marcelo Fabián Frias

Consejero de estudios: Dr. Carlos Gustavo López Pombo

Buenos Aires, 2013

MEJORAS A LA DEMOSTRACIÓN INTERACTIVA DE PROPIEDADES ALLOY UTILIZANDO *SAT-SOLVING*

El análisis formal de especificaciones de software suele atacarse desde dos enfoques, usualmente llamados: *liviano* y *pesado*. En el lado liviano encontramos lenguajes fáciles de aprender y utilizar junto con herramientas automáticas de análisis, pero de alcance parcial. El lado pesado nos ofrece lograr certeza absoluta, pero a costo de requerir usuarios altamente capacitados.

Un buen representante de los métodos livianos es lenguaje de modelado Alloy y su analizador automático: el Alloy Analyzer. Su análisis consiste en transcribir un modelo Alloy a una fórmula proposicional que luego se procesa utilizando *SAT-solvers* estándar.

Esta transcripción requiere que el usuario establezca cotas en los tamaños de los dominios modelados en la especificación. El análisis, entonces, es parcial, ya que está limitado por esas cotas. Por ello, puede pensarse que no es seguro utilizar el Alloy Analyzer para trabajar en el desarrollo de aplicaciones críticas donde se necesitan resultados concluyentes.

En esta tesis presentamos un cálculo basado en álgebras de Fork que permite realizar demostraciones en cálculo de secuentes sobre especificaciones Alloy. También hemos desarrollado una herramienta (Dynamite) que lo implementa. Dynamite es una extensión del sistema de demostración semiatomático PVS, método *pesado* ampliamente utilizado por la comunidad. Así, Dynamite consigue complementar el análisis parcial que ofrece Alloy, además de potenciar el esfuerzo realizado durante una demostración usando el Alloy Analyzer para detectar errores tempranamente, refinar secuentes y proponer términos para utilizar como testigos de propiedades cuantificadas existencialmente.

Palabras clave: Demostración de teoremas interactiva, SAT-Solving, cálculo para Alloy, PVS, Análisis de especificaciones de software.

IMPROVEMENTS TO INTERACTIVE THEOREM PROVING OF ALLOY PROPERTIES USING SAT-SOLVING

Formal analysis of software models can be undertaken following two approaches: the lightweight and the heavyweight. The former offers languages with simple syntax and semantics, supported by automatic analysis tools. Nevertheless, the analysis they perform is usually partial. The latter provides full confidence analysis of models, but often requires interaction from highly trained users.

Alloy is a good example of a lightweight method. Automatic analysis of Alloy models is supported by the Alloy Analyzer, a tool that translates an Alloy model to a propositional formula that is then analyzed using off-the-shelf SAT-solvers. The translation requires user-provided bounds on the sizes of data domains. The analysis is limited by the bounds, and is therefore partial. Thus, the Alloy Analyzer may not be appropriate for the analysis of critical applications where more conclusive results are necessary.

In this thesis we develop Dynamite, an extension of PVS that embeds a complete calculus for Alloy. PVS is a well-known heavyweight method. It provides a semi-automatic theorem prover for higher order logic. Dynamite complements the partial automatic analysis offered by the Alloy Analyzer with semi-automatic verification through theorem proving. It also improves the theorem proving experience by using the Alloy Analyzer to provide automatic functionality intended for early detection of errors, proof refinement and witness generation for existentially quantified properties.

Keywords: Interactive Theorem Proving, SAT-Solving, Alloy Calculus, PVS, Specification Analysis.

Agradecimientos / Acknowledgments

Gracias a todos. Los que están cerca, los que están lejos, los que ya no están. Si hoy puedo estar escribiendo estas líneas es porque ustedes estuvieron ahí.

Gracias a mis docentes, a mis alumnos, a mis compañeros, a mis colegas, a mis amigos. Gracias por hacerme sentir que lo único que separa esos grupos, son las comas. Tengo una dedicatoria que los nombra explícitamente a todos, pero no me alcanza ni el margen, ni la hoja.
Les prometo llevarla escrita en la mirada, en cada abrazo que nos reúna.

Thanks to Dr. Natarajan Shankar, Dr. Daniel Jackson, Dr. Santiago Figueira, and Dr. Carlos Areces for giving me the honor of having them as the committee of this thesis. Their questions, advices, and suggestions have resulted in a dramatic improvement in the quality of this document.

Gracias a mis queridos amigos que se vieron forzados a jugar de *sparrings idiomáticos* y tanto trabajo tuvieron revisando este documento: Nico y Elena, GG y M, Manu.

Gracias, Marcelo y Charles, por la dirección, por el apoyo (académico y emocional), por la ayuda incondicional y, por sobre todo, por brindarme su amistad.

Gracias a todas las ramas, hojas y hojitas de mi familia: GG, M, Apel, Albert, Beba, Alfredo, Silvia, César, Titi, Ricky, Rochy, Coy, Nelson, Ale, Coco, Gri, Miguel, Norman, Norma, Azul, Marta, Juan. Gracias por todo el amor, la fuerza, la paciencia. Nada hubiera podido hacer sin ustedes. Nada sería. Nada valdría la pena. Gracias por enseñarme a querer ser una mejor persona en cada momento.

Dedicatoria

A todos los que les agradecí.

En especial a Norma y a Azul. El amor que me dió la vida (en el sentido amplio) y el amor que me permite disfrutarla (en el sentido amplio).

Yo soy el faro, ahora. Nunca más les va a faltar la luz que yo pueda darles.

CONTENTS

1. INTRODUCTION

Abstractions are foundational parts of the software development process. As in many human activities, they are the first step of every significant software project. Experience has shown that misconceptions in the abstractions reveal themselves as major bugs in the implementation, and many times they are very hard to discover until it is too late. That is, until it is too expensive or even impossible to repair them. Thus, such misconceptions must be detected and corrected as soon as possible.

From simple and informal arrows and boxes diagrams to elaborated formal methods, the variety of conceptual tools used to document, communicate and analyze abstractions is vast and dynamic. Dynamic in the sense of their evolution, whose trail may be seen in the many techniques and artifacts that have appeared and have been left aside over the years.

Focusing on the analysis of abstractions, we may obtain better results as we move from informal to formal methods. Natural language and diagrammatic sketches with no formal semantics usually fit adequately to communicate basic ideas, but they are not a rich field to search for flaws.

If we just concentrate on the degree of automation offered by the formal methods aimed at the analysis of abstractions, the spectrum ranges from fully automatic to fully user-driven mechanisms. As examples of methods that lie on the former side of the spectrum we can mention Model Checking [Clarke et al. (1999)] using tools such as SPIN [Holzmann (2003)] and SAT-Solving using, for instance, the Alloy Analyzer [Jackson (2012)]. Such analysis methods require little effort on the part of the user, and are therefore usually called *lightweight* formal methods.

If we confine ourselves to the analysis technique of determining if a model (specification) is appropriate by analyzing whether certain given properties hold in the model, then both model checking and SAT-solving can be used to automatically check it. But full automation has its price, often paid by limitations on the kind of analysis provided, or by its lack of scalability. In particular, both SAT-solving and model checking have limitations on the expressiveness of the languages they can analyze. This is particularly clear in the case of SAT-solving, where the source language is propositional logic. Also, most model checkers support decidable fragments of logics.

Despite their limitations, automatic methods have qualities that make them valuable. Even if the model we are working on is expressed in a more expressive formalism, we could translate it to a propositional formula (in the case of SAT-solving) or to an automaton (in the case of model checking). The (probably weaker) model thus obtained can be automatically analyzed. Notice that the result of the analysis of the weaker model can offer partial information about the original model.

As an example, let us consider the Alloy modeling language [Jackson et al. (2001)]. Alloy is a formal specification language that allows one to create data domains, and express properties of relations defined over those domains. We will present Alloy in Section 2.1, but we point out here that Alloy is not decidable since it extends classical first-order logic. It even includes reflexive-transitive closure of binary relations, which is not expressible in classical first-order logic.

The Alloy Analyzer [Jackson (2012)] is a tool that allows one to automatically analyze Alloy models by searching for counterexamples for a given property using off-the-shelf SAT-solvers. There is a visible impedance mismatch between the undecidable Alloy language and the decidable language on which SAT-solvers operate. The gap is bridged by translating Alloy models to propositional models.

The translation does not come for free, it requires users to provide bounds (called *scopes* in the Alloy terminology) on the size of data domains. The bounded model is the one translated. The result of the SAT-based analysis is then valid for those semantic structures whose data domains are constrained by the chosen scopes, i.e., if a counterexample is not found, one might still exist if larger scopes were chosen. While this analysis technique has obvious limitations, it is nevertheless very useful when creating a model.

According to the *small scope hypothesis* [Andoni et al. (2002)], although possible, it is seldom the case that errors introduced in a model can only be exhibited within large models[1]. The aforementioned hypothesis even claims that most software errors can be made explicit by resorting to small domain sizes. Therefore, if we assume that this hypothesis holds, we may infer that many errors introduced when building a model can be discovered by performing bounded analysis using small bounds.

On the other side of the spectrum of tools, we may find the so-called *heavyweight* formal methods, named this way after the effort that tools and techniques in this group impose on the user. Tools based on interactive theorem provers, such as PVS [Owre et al. (1992)], Isabelle [Nipkow et al. (2002)] or Coq [Bertot et al. (2004)], are good examples of this kind of methods, which require a significant effort from the user. User guidance offers more conclusive analysis techniques, but requires trained users. Also, user-guided techniques are often time consuming.

The bright side of using heavyweight methods is that total analysis can be performed with their assistance even on specifications from languages as much expressive as Alloy, or even more expressive. For instance, there are many complete theorem provers for classical first-order logic. These include

[1] The discussion about the validity or not of the small scope hypothesis is beyond the objectives of this work. We point the interested reader to [Andoni et al. (2002)] and [Jackson and Damon (1996)], among others.

techniques for automatically proving “easy” properties, but more complex ones require creative steps that must be guided by a human user.

Proving a theorem can be a difficult and tedious activity, even more so if incorrect hypotheses are introduced along the way. This situation, which is more common than it is desirable, is both discouraging and time consuming. Notice that every proof step that depends on the wrong hypothesis has to be reconsidered when developing a new proof.

For instance, in order to prove in PVS that a collection of formulas $\Delta = \{ \delta_1, \ldots, \delta_m \}$ follows from a collection of hypotheses $\Gamma = \{ \gamma_1, \ldots, \gamma_k \}$, one begins with the *sequent* $\Gamma \vdash \Delta$. Applying inference rules, from $\Gamma \vdash \Delta$ one must reach other sequents that can be recognized as valid (for example, sequents of the form $\alpha \vdash \alpha$). The intuition behind the sequent $\Gamma \vdash \Delta$ is that, from the conjunction of the formulas in Γ, the disjunction of the formulas in Δ must follow. The formulas in Γ (Δ) are called the *antecedent* (*consequent*) of the sequent.

When an inference rule is applied on a sequent S, new sequents $S_1, \ldots, Sn$ are produced. Proving sequent S then reduces to finding proofs for the intermediate sequents $S_1, \ldots, Sn$. Our experience as users of PVS is that, when proving a given sequent, the number of antecedents and consequents in intermediate sequents tends to grow. This often leads to sequents containing formulas that are unnecessary for their proof. These formulas make identification of new proof steps more complex. PVS provides a command (`hide`) for hiding hypotheses and conclusions in sequents, yet its incorrect use may lead to hiding necessary antecedents or consequents, making the proof infeasible.

Even more, while dealing with propositional connectives is straightforward in calculi such as the one provided by PVS, quantifiers pose a challenge. A particularly complex problem is that of finding witnesses when attempting to prove existentially quantified assertions. In order to prove that a formula $\exists x : \alpha(x)$ holds, the user must present a term t, such that $\alpha(t)$ is always true.

Returning to Alloy, for many modeling situations the kind of analysis offered by the Alloy Analyzer is entirely satisfactory. In some cases, however, this analysis may fall short. This is evident when building models for critical systems. In those cases, knowing that no small errors exist in the model is not enough. Therefore, in this thesis we propose to extend the SAT-solving analysis provided by the Alloy Analyzer with a user-guided theorem prover based on PVS.

However, although this seems a significant contribution because it gives the user the possibility of reaching complete confidence in the Alloy model, the heavyweight task of using an interactive theorem prover threatens the usability of the tool. For that reason, we also developed automatic techniques, relying on the Alloy Analyzer, in order to lighten the workload of the task of proving.

Contributions

The contributions of this thesis can then be summarized as follows:

- We present a complete proof calculus for the Alloy modeling language.
- We introduce an interactive theorem prover for Alloy assertions that extends the PVS theorem prover so that Alloy assertions can be proved using Alloy syntax. This theorem prover, called Dynamite, was developed by us as part of the work of this thesis.
- We facilitate the interaction between PVS and the Alloy Analyzer in order to reduce the number of theorem proving errors induced by introduction of false lemmas, hiding of necessary hypotheses, or introduction of erroneous hypotheses.
- We present a heuristic to reduce the proof search space based on the use of UnSAT cores to remove potentially unnecessary antecedents and consequents in a sequent. The technique also allows us to remove formulas from the underlying theories being considered.
- We present a technique based on SAT for automatic generation of witness candidates for existentially quantified Alloy assertions.
- We present several examples, including some complex case-study [Ramananandro (2008); Zave (2005, 2006)], that allow us to assess the usefulness and usability of Dynamite.

These contributions have been presented in [Frias et al. (2007); Moscato et al. (2010, 2014)]. This document presents revised and augmented versions of the aforementioned contributions.

Organization

The thesis is organized as follows. In Chapter 2 we present a brief introduction to the Alloy modelling language, the Alloy Analyzer, PVS and the sequent calculus provided by it. In Chapter 3 we present the formal syntax and semantics of Alloy, the Fork Algebras we used as background logic for proving Alloy assertions, and the complete calculus for Alloy that will be made accessible through our tool, Dynamite. In Chapter 4 we describe the way in which the complete calculus presented in Chapter 3 is embedded in PVS, as well as discuss important implementation details. Also in Chapter 4, we describe the features of Dynamite. Chapter 5 is devoted to report the results of the use of Dynamite on several case studies. Some of the goals we envisioned when we started this work were previously addressed either by colleagues or ourselves. In Chapter 6 we present a brief review of such related work, and discuss some conclusions and further improvements to the tool.

2. MODELING AND ANALYSIS OF ABSTRACTIONS

When dealing with abstractions, their construction and analysis, the minimum-effort way to cope with these tasks is offered by lightweight methods. Languages with simple syntax and semantics, diagrammatic representations and fully automatic tool support, among other characteristics, make them appealing to a wide spectrum of the public. Nevertheless, the power of the analysis provided is, in most cases, not suited to critical developments.

On the other hand, heavyweight methods allow the user to perform complete analysis on the abstractions, but at the cost of losing the automation. Clever interactions at crucial points of the analysis process are often indispensable to fulfill the task. This makes the application of such methods an error-prone activity. Furthermore, besides requiring more user effort, they usually demand a high level of training, which in turn reduces the spectrum of public able and willing to use them.

This chapter is intended to contrast both approaches by showing representatives of each kind. First we will present Alloy as an example of a lightweight method. We will introduce rudiments about the language and its associated analysis tool: the Alloy Analyzer. The way in which the analyzer can be used as an assistant in the modeling process will be explained. Next, we will introduce the Prototype Verification System (PVS), an environment supporting the development and the analysis of specifications, based in theorem proving for classical higher-order logic. We will discuss the underlying formal machinery implemented by PVS: the sequent calculus.

2.1 An Introduction to Alloy

In this section we describe the Alloy modeling language and the Alloy Analyzer with the level of detail required in order to follow this work. For a thorough description we point the reader to the book "Software Abstractions: Logic, Language, and Analysis" [Jackson (2012)], which provides numerous examples of varied complexity illustrating features of Alloy. Nevertheless, a formal description of the syntax and semantics of the language will be presented in the next chapter.

2.1.1 Alloy terms and formulas

As with every formal language, Alloy can be studied from a syntactic or from a semantic point of view. On the semantic side, we will be working with relations of arbitrary, but finite, arity. These relations, which can be

```
sig Name { }
```

(a)

$$\mathfrak{a}_M(\mathsf{Name}) = \{\, \langle n_0 \rangle \,, \langle n_1 \rangle \,\}$$
$$\mathfrak{a}'_M(\mathsf{Name}) = \{\, \langle n_0 \rangle \,\}$$
$$\mathfrak{a}''_M(\mathsf{Name}) = \{\ \}$$

(b)

Fig. 2.1: *(a)* A signature declaration and *(b)* three different instances for it.

seen as sets of tuples of atomic elements, will be the first order citizens of the language.

The gap between the syntactic domain and the semantic domain is bridged using *instances*, also called *environments*. Much the same as valuations from classical first-order logic map variables to semantic values, environments map Alloy expressions to relations. Given an alloy model M, we denote by $\mathfrak{a}_M$ an instance for M.

Unary relations, along with the atoms that they contain, can be declared in Alloy by defining so-called *signatures*. For example, the statement shown in Fig. 2.1.a forces the existence of an unary relation $Name$ holding all the atoms of type $Name$ in the range of each instance applicable in the model M. In fact, the symbol Name can be used to denote that unary relation in any expression of the specification. Fig. 2.1.b shows the value taken by Name in three different and valid instances for the model M.

Alloy provides multiplicity keywords in order to restrict the relations introduced by a signature. These keywords are: some (the relation can not be empty), one (there can be only one tuple in the relation), lone (the relation can have at most one tuple). The table 2.1 shows which of the instances of Fig. 2.1.b are valid if each multiplicity keyword was added in turn to the declaration in Fig. 2.1.a.

Let us suppose we want to model a simple file system using Alloy. The main concepts are: directories can contain files and other directories, every element inside a directory can be referred to using a name that can vary in

	one sig Name { }	some sig Name { }	lone sig Name { }	sig Name { }
$\mathfrak{a}_M(\mathsf{Name}) = \{\, \langle n_0 \rangle \,, \langle n_1 \rangle \,\}$	✗	✓	✗	✓
$\mathfrak{a}'_M(\mathsf{Name}) = \{\, \langle n_0 \rangle \,\}$	✓	✓	✓	✓
$\mathfrak{a}''_M(\mathsf{Name}) = \{\ \}$	✗	✗	✓	✓

✓	valid
✗	invalid

Tab. 2.1: Signature multiplicity keywords example.

other directories, there is only one *root* directory, every non-root directory must have a parent, i.e. the directory containing it.

The directories could be represented by means of a signature Dir, as we did it with the names previously. Now we have a distinguished directory: the root. To embody this feature, Alloy allows us to establish inheritance relationships between signatures. Then,

```
one sig Root extends Dir { }
```

declares a *Root* signature included in *Dir*. Notice that, because of the one keyword, *Root* is a singleton in every instance.

To model the relationship between a directory and its parent, we could use a binary relation $parent \subseteq Dir \times Dir$. Relations of arity greater than one are defined by declaring so-called *fields* in a signature. For example, field *parent* in the signature *Dir*, shown below, denotes a binary relation included in the cartesian product $Dir \times Dir$.

```
sig Dir { parent: Dir }
```

In fact, that declaration states an extra restriction on the tuples of *parent*: for every atom $d \in Dir$, there must be one, and only one, tuple in *parent* with d as its first element. Alloy allows us to include the multiplicity keywords mentioned before in order to relax this constraint.

In our example, since every directory except for root must have one and only one parent, we need that for every $d \in Dir$ there be at most one tuple in *parent* beginning with d. This restriction can be stated with the lone keyword.

```
sig Dir { parent: lone Dir }
```

In addition to lone, one and some, previously presented, the keyword set may be used to state no restriction at all on the tuples of *parent*.

Now, once the *parent* relation has been defined, we could write an Alloy expression that relates every directory with its "grandparent", if it has one. The operator for relational composition ".", called *navigation* in Alloy terminology, allows us to write that expression as parent.parent.

As described before, a directory can contain files and directories. Naturally by now (we hope) the files will be represented as a new signature File. So, using signature inheritance one can group directories and files in an abstract concept. Let's call it *object*.

```
abstract sig Object {}
sig File extends Object {}
```

The keyword abstract states that *Object* will only hold elements from inheriting signatures.

$$\begin{aligned}\mathfrak{a}_M(\mathsf{d}) &= \{\langle d_0\rangle\}\\ \mathfrak{a}_M(\mathsf{contents}) &= \{\langle d_0, n_0, d_0\rangle, \langle d_0, n_0, f_0\rangle, \langle d_1, n_0, d_0\rangle\}\\ \mathfrak{a}_M(\mathsf{d.contents}) &= \{\langle n_0, d_0\rangle, \langle n_0, f_0\rangle\}\end{aligned}$$

Fig. 2.2: Example of navigation between non binary relations.

Recalling that every object must have a name local to the directory containing it, the contents of a directory can be represented as a ternary relation $contents \subseteq Dir \times Name \times Object$. In Alloy, relations of arity greater than 2 are also declared as fields, but using the arrow operator "−>".

```
sig Dir extends Object {
    contents:  Name −> Object,
    parent:  lone Dir
}
```

$$\begin{aligned}contents = \{&\langle d_0, n_0, d_0\rangle,\\ &\langle d_0, n_0, f_0\rangle,\\ &\langle d_1, n_0, d_0\rangle\}\end{aligned}$$

Every tuple $\langle d, n, o\rangle \in contents$ means that o is contained in[1] d, under the name n. Along the last showed declaration of Dir, a possible instance for it is depicted. There, d_0 and d_1 are atoms from Dir, n_0 from $Name$ and f_0 from $File$.

Lets assume we have an Alloy expression d denoting a singleton unary relation that holds the atom d_0. We can express the collection of objects contained in d_0, along with the respective name of each one of them in d_0, by using the composition operator in the following way: d.contents. Composition of binary relations is well understood but, for relations of higher arity, the following definition for the composition of relations is considered:

$$R.S = \{\langle a_1, \ldots, a_{i-1}, b_2, \ldots, b_j\rangle : \\ \exists b\,(\langle a_1, \ldots, a_{i-1}, b\rangle \in R \;\wedge\; \langle b, b_2, \ldots, b_j\rangle \in S)\}\ .$$

being R and S relations of arity i and j respectively. Then, the expression d.contents denotes a binary relation where in every tuple can be found a directory contained in d and the name for which it can be referenced in d (see Fig. 2.2).

In general, Alloy terms are built from signatures, signature fields and constants such as **univ** (set of all objects in the model), **iden** (identity binary relation on the set **univ**), and **none** (the empty set). Relational operators

[1] Notice that the phrase "o is contained in d" should not be read literally. As mentioned above, since both o and d are atoms they do not contain anything, they are indivisible. That phrase is intended to refer to the conceptual model being described by the Alloy specification. Thus, it should be read as "*the object represented by* o is contained in *the directory represented by* d".

are used to build more complex terms. Difference, union, intersection and composition of relations are denoted by $-$, $+$, & and $\cdot$, respectively. The symbol $\sim$ denotes the transpose of a binary relation. This operation, also known as *inverse*, flips around the elements in each pair of a binary relation. Transitive closure and reflexive-transitive closure of binary relations are denoted by ^ and $*$, respectively. For example, denoting d a singleton in Dir, the term d.^parent denotes the set containing the parent of d, the parent of its parent, and so on.

If we return to the figure 2.2 we may see an anomaly of the model: d_0 contains two different objects with the same name n_0. In order to avoid such instances, we can add multiplicity keywords to the contents definition. The following declaration

```
sig Dir extends Object {
    contents: Name −> lone Object,
    parent: lone Dir
}
```

states that for every name n in $Name$, there may be at most one tuple in $\mathfrak{a}_M$(d.contents) with n as its first element, avoiding the instance showed in Fig. 2.2.

If we turn now to the parent field, we already constrained it to allow only one parent per directory. But it remains to be said that every directory that is not the root must have a parent and that the root must not have it.

To impose this kind of restrictions, we can add formulas to the model and mark them as axioms, called *facts* in Alloy terminology. To construct formulas, Alloy provides: unary cardinality testing predicates (such as the ones explained for signatures: being e an Alloy expression one e, some e, lone e and no e, for testing of emptiness), equality and inclusion between expressions (= and in, respectively), all the usual boolean connectives (! for negation, && for conjunction, || for disjunction, => for implication, <=> for logical equivalence), and several flavors of quantifications.

The following facts express the aforementioned constrains.

```
fact RootHasNoParent { all r: Root | no r.parent }
fact OneParent { all d: Dir − Root | one d.parent }
```

As expected, the RootHasNoParent axiom says that for all root directories r, the set (unary relation, in fact) of its parents is empty. Alloy allows us to express restrictions on the range of the quantified variables as if it were the declaration of a field. So, the variable d of the OneParent fact ranges over all directories but the root. For all such directories, the fact assures that they have exactly one parent each.

We may also add hypotheses about the model and test their validity. These hypotheses are called *assertions* in Alloy. For example, the assertion

NoDirAliases, shown below, states that every directory may be contained in at most one other directory.

```
assert NoDirAliases { all o: Dir | lone (contents.o) }
```

Note that while o.contents represents all the objects contained in the directory o (along with each respective name), the expression contents.o denotes all the directories which include o (along with the name by which it is known in each of them).

Alloy also allows the user to define custom predicates and functions. They act as named formulas and expressions that can be reused across the specification. For example, the following axiom ensures that the directory hierarchy has no loops.

```
fun ancestors[x: Dir]: Dir { x.^parent }

fact NoOwnAncestor { all d: Dir | d !in ancestors[d] }
```

The Alloy language provides support for integer numbers. But, in order to be able to automatically analyze models, the user must provide a bound to the count of integers being referenced in the specification. This bound, called the *bit width*, is the number of binary digits used to represent integer atoms using 2's complement arithmetic. For instance, with 5 as bit width, we can represent integers -16 through 15.

There is a built-in signature **Int** that denotes the set of all integers (up to the user provided bound). So, integers can appear as atoms in relations. For example, if we wanted to enrich the directories of our example with the count of elements contained by them, we could write:

```
sig DetailedDir extends Dir { filesCount: Int }
```

Alloy offers arithmetic operators for addition (add), substraction (sub), multiplication (mul), integer division (div) and remainder of an integer division (rem) as well as the usual operators for comparison between integers $(=, <, <=, >, >=)$. All these operators are defined using 2's complement arithmetic. For example, $15 + 1 = -16$ holds when bit width 5 is chosen. The latest version of Alloy includes a treatment of integers that avoids some of the overflow issues that the user may face when dealing with such a bounded representation of numbers.

Nevertheless, these operations may not be applied on sets of integers, not even singletons, but on integer numbers themselves. Then, Alloy provides a built-in function, called **int**, to cast from sets of integers to integers. So, the addition between the count of files in directories d1 and d2 may be denoted by the expression

```
int[d1.filesCount] add int[d2.filesCount]
```

When applied to a relation R with cardinality greater than a singleton, **int**[R] denotes the result of the sum of every numeric atom in R.

In order to force the field filesCount in the example to hold the count of elements contained by every directory, we may use the Alloy operator #, that returns the cardinality of its argument. Thus, the mentioned constraint may be stated as the following fact.

```
fact { all d: DetailedDir | int[d.filesCount] = #(d.contents) }
```

The backward cast, i.e. from integer numbers to integer atom singletons, is performed by the built-in function **Int**. So, the previous fact could be written as shown below.

```
fact { all d: DetailedDir | d.filesCount = Int[#(d.contents)] }
```

2.1.2 The Alloy Analyzer

One of the more appealing features of Alloy is the support for validation given by the Alloy Analyzer. Conceptually, the Alloy Analyzer may be seen as an instance builder. Given an Alloy model and a *scope specification* (basically bounds for the sizes of the signatures), the analyzer tries to automatically build some instance of such size in which all the facts of the model hold. We will say that such instance is a *legal instance of the model*. The whole search space denoted by the scope specification is exhaustively traversed so that, if any instance can be build under the given settings, the analyzer will do it.

In order to perform this task, the Alloy Analyzer translates the model to a propositional formula. The translation heavily depends on the bounds declared in the `check` command. Once a propositional formula has been obtained, the Alloy Analyzer employs off-the-shelf SAT-solvers, and in case a model of the formula is obtained, it is translated back to an instance of the source model and presented to the user using different visualization algorithms.

Simulation. Recall that, in the previous section, we exemplified how a flaw in a model can be detected by inspecting a specific instance. The Alloy Analyzer allows the user to exercise this simulation strategy in order to iteratively improve the specification being developed.

The language itself provides particular-purpose constructions to specify instance-building requests and the bounds to be used on it. For example, the following statement specifies a search for an instance in which all the signatures may have at most three elements, but the directory signature is forced to have exactly two:

```
run {} for 3 but exactly 2 Dir
```

It is worth noting that this simulation-assisted modeling technique can only be carried out if the analyzer succeeds in the construction of an instance for the given model and scope. Complementarily, the fail of the analyzer can be understood as an indication of inconsistency of the model. Nevertheless, this evidence is not conclusive because it may be the case that an instance could be built if the scope is increased.

Testing of assertions. Besides the instance building feature, the Alloy Analyzer may be used to test the conjectures stated as assertions. This test is done by searching for counterexamples of the assertion. In fact, this search is another side of the instance builder feature explained earlier. The Alloy Analyzer tries to build an instance for which all the facts hold, but the assertion does not.

Again, one may include specifications of these analyses in the model itself. The following code determines a search for counterexamples of the assertion NoDirAliases with the same scope of the *run* mentioned before.

```
check NoDirAliases for 3 but exacty 2 Dir
```

As in the *run* case, if a counterexample cannot be generated by the analyzer with the provided scopes, it does not mean that no counterexample exists. Perhaps an increasing of the scopes is needed in order to fulfil the counterexample construction. But it may be the case that no counterexample could be made, no matter how big the scopes would be set. So, even though the Alloy Analyzer is so useful to develop a consistent specification without errors that could be exposed by (generally) small examples, it cannot give absolute certainty on the correctness of the specification.

2.2 A Step Further: Heavyweight Methods

In some cases, such as the development of safety-critical systems, the partial assurance provided by lightweight methods is not enough, and more sophisticated and complete methods must be used. These methods are called heavyweight because they allow one to achieve full confidence in a model, but usually require expertise and specific training from the user; these, many times, discourages their use.

Semiautomatic theorem provers are good examples of heavyweight formal methods, and they are one of the few available choices when dealing with an undecidable logic, such as Alloy. One of the more prominent exponents of this family of tools is the Prototype Verification System (PVS) [Owre et al. (1992)].

2.2.1 The Prototype Verification System

The Prototype Verification System (PVS) provides an automated environment aimed at the development and verification of complex models. Its features make it suitable for a wide variety of application domains, where it has been used. A comprehensive, but perhaps not fully up-to-date as of the writing of this document, list of publications about the application of PVS to different fields and topics may be found in [Rushby (2000)].

Some of the more relevant characteristics of PVS are the specification language and the semiautomatic theorem prover that serves as the main tool for verification of the models. The PVS specification language is based on classical typed higher-order logic. Besides the built-in available types (such as booleans, integers, reals, etc.) the user may define new uninterpreted basic types as well as complex types using constructors for functions, tuples, sets, enumerations, etc. Since the type system defined by the user is not forced to be decidable, the typechecking process may return obligations that can be proven using the PVS prover. In fact, this feature gives much more expressiveness to the models.

PVS's theorem prover is a framework that allows the user to perform proofs using sequent calculus. Since some of the highlights of Dynamite are the automatic help offered in key steps of the proof, in the next section we will explain basic concepts about the calculus. PVS's support for user-defined proof tactics allows us to define specific rules to take advantage of some aspects of the Alloy logic.

A comprehensive library of PVS theories covering diverse topics provides wide support to the application of PVS in different areas.

2.2.2 Sequent Calculus

We already introduced the notion of *sequent* in the introduction of this work (chapter 1). Recall that a sequent is simply a pair of finite sequences of formulas. Usually the formulas in the first list of the pair act as hypothesis and the formulas in the second one act as thesis. These sequences are respectively called *antecedent* (usually denoted by Γ) and *consequent* (Σ).

In the brief description given in the introduction, it may be noted that the proofs in a sequent calculus can be seen as a tree. When the user starts a proof, the proof tree has only one node. This tree will grow as the result of the application of the so called *proof rules*. These rules are traditionally depicted as:

$$\frac{\Gamma_1 \vdash \Delta_1 \qquad \ldots \qquad \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}\ (\mathbf{R})$$

which states that the application of the rule **R** on a sequent with form $\Gamma \vdash \Delta$ results in n new sequents $\Gamma_1 \vdash \Delta_1$ to $\Gamma_n \vdash \Delta_n$.

$$\frac{\Gamma \vdash \alpha, \beta, \Delta}{\Gamma \vdash (\alpha \vee \beta), \Delta}\,(\vdash\vee) \qquad \frac{\Gamma, \alpha \vdash \Delta \qquad \Gamma, \beta \vdash \Delta}{\Gamma, (\alpha \vee \beta) \vdash \Delta}\,(\vee\vdash) \qquad \frac{\Gamma \vdash \alpha, \Delta}{\Gamma, (\neg\alpha) \vdash \Delta}\,(\neg\vdash)$$

$$\frac{\Gamma \vdash \alpha, \Delta \qquad \Gamma \vdash \beta, \Delta}{\Gamma \vdash (\alpha \wedge \beta), \Delta}\,(\vdash\wedge) \qquad \frac{\Gamma, \alpha, \beta \vdash \Delta}{\Gamma, (\alpha \wedge \beta) \vdash \Delta}\,(\wedge\vdash) \qquad \frac{\Gamma, \alpha \vdash \Delta}{\Gamma \vdash (\neg\alpha), \Delta}\,(\vdash\neg)$$

$$\frac{\Gamma, \alpha \vdash \beta, \Delta}{\Gamma \vdash (\alpha \Rightarrow \beta), \Delta}\,(\vdash\Rightarrow) \qquad \frac{\Gamma \vdash \alpha, \Delta \qquad \Gamma, \beta \vdash \Delta}{\Gamma, (\alpha \Rightarrow \beta) \vdash \Delta}\,(\Rightarrow\vdash)$$

$$\frac{\Gamma, \alpha\{x \leftarrow t\} \vdash \Delta}{\Gamma, (\forall x : \alpha) \vdash \Delta}\,(\forall\vdash) \qquad \frac{\Gamma \vdash \alpha\{x \leftarrow a\}, \Delta}{\Gamma \vdash (\forall x : \alpha), \Delta}\,(\vdash\forall)$$

$$\frac{\Gamma, \alpha\{x \leftarrow a\} \vdash \Delta}{\Gamma, (\exists x : \alpha) \vdash \Delta}\,(\exists\vdash) \qquad \frac{\Gamma \vdash \alpha\{x \leftarrow t\}, \Delta}{\Gamma \vdash (\exists x : \alpha), \Delta}\,(\vdash\exists)$$

Where t is a term, a is a fresh constant, and $\alpha\{x \leftarrow t\}$ is the formula resulting from substituting all free occurrence of the variable x in α with t.

$$\frac{\Gamma, \alpha \vdash \Delta \qquad \Gamma \vdash \alpha, \Delta}{\Gamma \vdash \Delta}\,(\text{Cut}) \qquad \frac{\Gamma_1, \alpha, \alpha, \Gamma_2 \vdash \Delta}{\Gamma_1, \alpha, \Gamma_2 \vdash \Delta}\,(\mathbf{C}\vdash) \qquad \frac{\Gamma \vdash \Delta_1, \alpha, \alpha, \Delta_2}{\Gamma \vdash \Delta_1, \alpha, \Delta_2}\,(\vdash\mathbf{C})$$

$$\frac{\Gamma' \vdash \Delta'}{\Gamma \vdash \Delta}\,(\mathbf{W}) \quad \text{where: } \Delta' \subseteq \Delta,\ \Gamma' \subseteq \Gamma \qquad \frac{}{\Gamma \vdash \Delta}\,(\text{Ax}) \quad \text{where: } \Delta \cap \Gamma \neq \emptyset$$

$$\frac{\Gamma_1, \alpha, \beta, \Gamma_2 \vdash \Delta}{\Gamma_1, \beta, \alpha, \Gamma_2 \vdash \Delta}\,(\mathbf{X}\vdash) \qquad \frac{\Gamma \vdash \Delta_1, \alpha, \beta, \Delta_2}{\Gamma \vdash \Delta_1, \beta, \alpha, \Delta_2}\,(\vdash\mathbf{X})$$

Fig. 2.3: Proof rules of sequent calculus.

Consider for instance the proof rules for conjunction $\wedge\vdash$ and $\vdash\wedge$ depicted in Fig. 2.3. Rule $\wedge\vdash$ shows that proving a sequent that contains the conjunction $\alpha \wedge \beta$ in the antecedent reduces to proving a sequent with both α and β in the antecedent (in this case, a single new sequent has to be proved). Similarly, rule $\vdash\wedge$ shows that proving a sequent with $\alpha \wedge \beta$ in the consequent reduces to proving two different sequents: one with α in the consequent, and another one with β in the consequent.

Besides the rules for conjunction, Fig. 2.3 shows examples of rules for standard boolean connectives, quantifiers, *structural rules* intended to reorder the formulas in the sequent ($\mathbf{C}\vdash$, $\vdash\mathbf{C}$, $\mathbf{X}\vdash$, $\vdash\mathbf{X}$, $\mathbf{W}$) and the *Cut* rule, which can be seen as the formalization of the introduction of some new hypothesis or as a case separation. PVS provides support for a calculus such as this. For details in how the proof rules are represented in PVS the reader is pointed to [Owre et al. (2001b)].

2.2.3 Critical Points of Sequent Calculus Proofs

A closer look at the proof calculus shows that most of the rules in Fig. 2.3 can be applied mechanically, as in the case of the rules for disjunctive connectives explained earlier. Nevertheless, there are some rules in which creativity is required. We say that the applications of these rules are *critical points* because they materialize the most error-prone moments of the proof. Such rules are the Cut rule, where an appropriate formula α has to be determined, and the rules for quantifiers.

There are theoretical results that would allow us to leave aside the cut rule. The Cut-elimination theorem [Gentzen (1935)] states that proofs can be replaced by (usually more complex) proofs that do not use this rule. But it is of great practical significance in the task of performing a proof using sequent calculus. For example, when attempting to prove a sequent $\Gamma \vdash \Delta$, many times a new hypothesis α is introduced as a means to simplify and modularize the proof. In the left new node after the application of the rule ($\Gamma, \alpha \vdash \Delta$) hypothesis α can be used in the proof of the sequent. But on the other hand, starting from the right new node ($\Gamma \vdash \alpha, \Delta$), it will have to be proved that α follows from some formulas in Γ and possibly the negation of some other formulas in Δ. This is, thus, a *critical point* of the proof, because of the loss of time the user could suffer if, after finishing the proof of the left subtree with the aid of formula α, it turns out that the new hypothesis cannot be discharged from Γ and Δ, making the previous proof effort useless.

In case we are confronted with a sequent with either an universally quantified formula in the consequent, or an existentially quantified formula in the antecedent, the quantifiers can be skolemized away through the application of proof rules $\vdash\forall$ and $\exists\vdash$, respectively. The other two rules involving quantifiers, $\vdash\exists$ and $\forall\vdash$, are intended to deal with the so called *existential strength quantifiers*. Both rules require the presentation of a term that can be used to replace the quantified variable. We will call it a *witness* of the validity of the property. We say that the application of these rules is also a critical point because the user has to provide the witness in order to complete the proof.

Notice that these two critical points are applicable to any sequent calculus with rules akin to the ones in Fig. 2.3. Nevertheless, every logic has its own features, which can expose more critical points in the proof process. In fact, we will see aspects specific to the calculus we propose to verify Alloy assertions in Section 4.2.

2.3 Towards a Symbiotic Approach

Jackson says about the use of heavyweight methods in the analysis of software abstractions [Jackson (2012)]:

> Completing a proof with the aid of a theorem prover usually demands an effort an order of magnitude greater than the modeling effort that preceded it, so for most applications, it's not cost-effective. For checking safety-critical abstractions, however, the additional assurance obtained from proof may be worthwhile.

We wanted Dynamite to be more than just a tradeoff between both approaches, the heavy one and the light one. We were aiming for a tool with the strength of a heavyweight method, but with simple syntax and semantics in order to fit the necessities of (as much as possible) software developments. We wanted to bring complete accuracy but reducing the cost of applying this technique, so that it can be applied in many more situations. Additionally, we intended to establish a synergistic relationship between tools of both worlds, so that the human effort to develop a proof could be supported in the critical steps by automated help.

In the next chapters we will describe Dynamite in detail. First, we will present the formal foundations of the tool. Then, we will follow with the implementation description and the explanation of prover helper features included in Dynamite.

3. FORMAL BACKGROUND

One of the original and main objectives of this thesis was to provide a mechanism to reach full confidence in the analysis of Alloy specifications. In particular, by offering a mechanism to ensure that an assertion is valid for all possible scopes. One way to reach this objective is to provide Alloy with a complete calculus. That calculus would allow one to prove that a given assertion can be deduced from the facts of the specification it comes from, concluding that it is valid.

To reduce the amount of work needed to give Alloy a complete calculus, we focus our effort in trying to reuse the calculus of a more powerful formalism. Such a formalism had to be able of mimic the semantics of every possible Alloy specification and had to have its own complete calculus, so that the theorems of that formalism state valid properties on the semantic Alloy models. Once such a formalism was selected, we had to design a semantic-preserving translation from Alloy to that formalism.

This chapter is intended to show that the aforementioned approach is correct. First, we will exhibit the syntax and formal semantics of Alloy. The notion of validity of an Alloy assertion will be formally stated as well. Then, we will focus on the target of the mentioned translation: the class of "proper point-dense omega closure fork algebras" (PDOCFA). The definition of this class will be given along with the presentation of a complete calculus for it. Later, we will define the translation $\mathbf{F}$ from Alloy models to PDOCFA theories. Finally, we will prove that any theorem in the PDOCFA theory resulting of the translation via $\mathbf{F}$ of some Alloy model is valid in it, and vice versa (*interpretability theorem*).

3.1 Semantics and Syntax of Alloy

Signature and field symbols will be called *relational variables*, whilst variables bound by quantifiers will be called *binding variables.* We will denote by *Names* the set of both variable symbols. Meanwhile the collection of all relational values (sets of tuples of atomic values) used to interpret Alloy specifications will be denoted by *Rel* .

There are two kinds of atomic values: uninterpreted atoms ($\mathcal{A}$) and numeric atoms ($\mathcal{Z}$). They only differ in the existence of a bijection (named *intVal*) from the latter to the set of integer numbers.

An *Alloy instance*, or *Alloy environment*, is a mapping

$$\mathfrak{i} : \mathit{Names} \to \mathit{Rel}$$

that assigns meaning to variable symbols. We denote by $\mathcal{I}$ the collection of all instances.

Given an Alloy expression expr and an Alloy instance $\mathfrak{i}$, the function E gives meaning to expr based on $\mathfrak{i}$. Correspondingly, given an Alloy formula α and an Alloy instance $\mathfrak{i}$, we will define a metapredicate that will indicate whether the formula α is true or false with respect to the instance $\mathfrak{i}$ (noted as $\mathfrak{i} \models \alpha$). Both E and $\models$ will be formally defined in the next section.

Notice that an instance does not need to be total, but we restrict the domain of the instances with respect to a given specification in the next definition.

Definition 3.1.1: (Legal instances of a specification) The collection of *legal instances of a given Alloy specification* S (denoted by $\mathcal{I}_{\mathsf{S}}$) contains every instance $\mathfrak{i}$ such that:

a) every constant symbol defined in S belongs to the domain of $\mathfrak{i}$.

b) there exists an $n \in \mathbb{N}$ s.t. all the integer numbers representable in a 2's complement representation of n bits are defined in $\mathfrak{i}$; more formally:

$$\mathfrak{i}(j) = \{\langle intVal^{-1}(j)\rangle\} \quad \text{for each integer } j \in [-2^{n-1}, 2^{n-1} - 1]$$

c) $\mathfrak{i} \models \alpha$ is true for every fact α in S.

Definition 3.1.2: (Consistence) An Alloy specification S is *consistent* if $\mathcal{I}_{\mathsf{S}}$ is not empty. Correspondingly, S is *inconsistent* if $\mathcal{I}_{\mathsf{S}}$ is empty.

Definition 3.1.3: (Counterexample) Given an Alloy specification S, an instance $\mathfrak{i} \in \mathcal{I}_{\mathsf{S}}$ is a *counterexample for an assertion* α *in* S if $\mathfrak{i} \models \alpha$ is false.

The rest of this section is devoted to give the formal details of the definitions just presented. We will begin with the function E and the metapredicate $\models$ and then we will focus on how the declarations constrain the instances.

Expressions and Formulas

Although in Section 2.1 we already gave an informal description of the main characteristics of the Alloy language, we present here in detail its syntax and semantics, as presented in [Jackson (2012)].

Diagram 3.1 shows the grammar for Alloy expressions. We denote by $\mathscr{E}^{\mathsf{Alloy}}$ the collection of expressions generated by such grammar. Just a few relational operators were not already presented in Chapter 2. They are: the domain restriction <: (that behaves as a filter on the tuples of a relation, keeping only those which have as a first component an element from a given set), the range restriction :> (similar to the aforementioned operation, but taking into account the last component of the tuples instead of the first), and the relational override ++ (that allows to update a relation using the

```
⟨expression⟩ ::= none | univ | iden | @ ⟨field⟩ | ⟨rel⟩ | ⟨var⟩
  |  [ # | ~ | * | ^ ] ⟨expression⟩
  |  ⟨expression⟩ [ + | & | - | ++ | <: | :> | . | -> ] ⟨expression⟩
  |  ⟨expression⟩ [ ⟨expression⟩,* ]
  |  ⟨formula⟩ [=>|implies] ⟨expression⟩ else ⟨expression⟩
  |  let ⟨let-binding⟩,+ | ⟨expression⟩
  |  { ([disj] ⟨var⟩,+ : ⟨expression⟩),+ [| ⟨formula⟩ | ⟨conj-block⟩] }
  |  Int | Int[⟨numeric-expr⟩]
  |  ( ⟨expression⟩ )

⟨let-binding⟩ ::= ⟨name⟩ = [ ⟨expression⟩ | ⟨formula⟩ | ⟨numeric-expr⟩ ]

⟨conj-block⟩ ::= {⟨formula⟩*}

⟨rel⟩ := ⟨sig⟩ | ⟨field⟩

⟨sig⟩ ::= [this | ⟨identifier⟩] [/ ⟨identifier⟩]*

⟨field⟩ ::= [this | ⟨identifier⟩] [/ ⟨identifier⟩]*

⟨var⟩ ::= ⟨identifier⟩

⟨identifier⟩ ::= [a-z]+[[a-z][0-9]_'"]*


⟨numeric-expr⟩ ::= ⟨number⟩
  |  - ⟨numeric-expr⟩
  |  # ⟨expression⟩
  |  ⟨expression⟩.sum | int[⟨expression⟩]
  |  ⟨numeric-expr⟩ [ add | sub | mul | div | rem ] ⟨numeric-expr⟩
  |  sum ⟨bindings⟩ | ⟨numeric-expr⟩
  |  ⟨var⟩
  |  let ⟨let-binding⟩,+ | ⟨numeric-expr⟩

⟨number⟩ ::= [0–9]+
```

Grammar diagram 3.1: Alloy expression grammar tuples of another relation). Alloy also supports more complex syntactic structures, such as macro expansions (using the let construct), *if-then-else*, and comprehension expressions.

Previous to the definition of the function E, which gives meaning to Alloy expressions, we state the notation we use in this document to represent well known relational operations.

Definition 3.1.4: (Relational operations) Being R_1 and R_2 relations of arbitrary but the same arity (say n), t a tuple of n elements, and A a relation or arity 1, we define:

Set-union	$R_1 \cup R_2 = \{t \mid t \in R_1 \text{ or } t \in R_2\}$
Set-intersection	$R_1 \cap R_2 = \{t \mid t \in R_1 \text{ and } t \in R_2\}$
Set-difference	$R_1 \setminus R_2 = \{t \mid t \in R_1 \text{ and } t \notin R_2\}$
Cardinality	$\#R_1 = \text{count of tuples in } R_1$
Projection of a tuple	$\Pi_i(\langle a_1, \cdots, a_i, \cdots, a_n\rangle) = a_i$
Domain of a relation	$dom(R_1) = \{a \mid \exists t \in R_1 \text{ and } \Pi_1(t) = a\}$
Range of a relation	$ran(R_1) = \{b \mid \exists t \in R_1 \text{ and } \Pi_n(t) = b\}$
Domain restriction	$A \lhd R_1 = \{t \mid t \in R_1 \text{ and } \Pi_1(t) \in A\}$
Range restriction	$R_1 \rhd A = \{t \mid t \in R_1 \text{ and } \Pi_n(t) \in A\}$
Relational override	$R_1 \oplus R_2 = R_2 \cup ((dom(R_1) - dom(R_2)) \lhd R_1)$
Identity	$iden_A = \{\langle a, a\rangle : a \in A\}$

Being R a n-ary relation and S a m-ary relation:

Cartesian product $R_1 \times R_2 =$

$$\{\langle a_1, \cdots, a_n, b_1, \cdots, b_m\rangle \mid \langle a_1, \cdots, a_n\rangle \in R_1 \text{ and } \langle b_1, \cdots, b_m\rangle \in R_2\}$$

Additionally, if R is a *binary* relation s.t. $R \subseteq A \times A$, we define:

Transpose $\sim R = \{\langle b, a\rangle \mid \langle a, b\rangle \ \in R\}$

Compositional closures

$$R^* = \text{smallest } T \text{ s.t. } iden_A \subseteq T \wedge T.T \subseteq T \wedge R \subseteq T$$
$$R^+ = R.R^*$$

Definition 3.1.5: (Meaning of Alloy expressions) The function

$$E : \mathscr{E}^{\mathsf{Alloy}} \to \mathcal{I} \to Rel$$

is defined as shown in tables 3.1 and 3.2.

Once defined the syntax and semantics of Alloy expressions, we can now proceed to formally define the grammar of Alloy formulas and then turn to their meaning. The Alloy formula grammar is depicted in Diagram 3.2.

e	$E(e)\mathfrak{i}$
none	$\emptyset$
univ	$\{\langle a\rangle \mid \exists\ S \in Ran(\mathfrak{i}) \text{ s.t. } \langle a\rangle \in S\}$
iden	$\{\langle a,a\rangle \mid \langle a\rangle \in E(\mathsf{univ})\mathfrak{i}\}$
Int	$\{\ \langle x\rangle \mid \langle x\rangle \in E(\mathsf{univ})\mathfrak{i} \wedge x \in \mathcal{Z}\}$
$\mathsf{Int}[n]$	$\{\langle intVal^{-1}(\mathfrak{cal}(n)\mathfrak{i})\rangle\}$
v	$\mathfrak{i}(v)$
c	$\mathfrak{i}(c)$
$\sim e$	$\sim E(e)\mathfrak{i}$
$\hat{}e$	$(E(e)\mathfrak{i})^{+}$
$*e$	$(E(e)\mathfrak{i})^{*}$
$e_1\ \&\ e_2$	$E(e_1)\mathfrak{i} \cap E(e_2)\mathfrak{i}$
$e_1 + e_2$	$E(e_1)\mathfrak{i} \cup E(e_2)\mathfrak{i}$
$e_1 - e_2$	$E(e_1)\mathfrak{i} \setminus E(e_2)\mathfrak{i}$
$e_1\ .\ e_2$	$E(e_1)\mathfrak{i}.E(e_2)\mathfrak{i}$
e_1 <: e_2	$E(e_1)\mathfrak{i} \lhd E(e_2)\mathfrak{i}$
e_1 :> e_2	$E(e_1)\mathfrak{i} \rhd E(e_2)\mathfrak{i}$
e_1 ++ e_2	$E(e_1)\mathfrak{i} \oplus E(e_2)\mathfrak{i}$
$e_1\ m_1$->$m_2\ e_2$	$E(e_1)\mathfrak{i} \times E(e_2)\mathfrak{i}$ (m_1 and m_2 are ignored)
$e[e_1, \cdots, e_n]$	$\begin{cases} E(e_f)\left(\mathfrak{i} \oplus \bigcup_{i=1}^{k}\{x_i \mapsto E(d_i)\mathfrak{i}\}\right) & \text{if } e \text{ is the function:} \\ & \mathsf{fun}\ f[x_1 : d_1, \cdots, x_k{:}d_k] : d_f\{e_f\} \\ E(e)\mathfrak{i}.E(e_1)\mathfrak{i}\ . \cdots .\ E(e_n)\mathfrak{i} & \text{otherwise.} \end{cases}$
$\mathsf{let}\ v = e_1 \mid e$	$E(e)(\mathfrak{i} \oplus \{v \mapsto E(e_1)\mathfrak{i}\})$
$\mathsf{let}\ v = n \mid e$	$E(e)(\mathfrak{i} \oplus \{v \mapsto \{\langle intVal^{-1}(\mathfrak{cal}(n)\mathfrak{i})\rangle\}\})$
$\mathsf{let}\ v = \alpha \mid e$	$\begin{cases} E(e)(\mathfrak{i} \oplus \{v \mapsto \emptyset\}) & \text{if } \mathfrak{i} \models \alpha. \\ E(e)\mathfrak{i} & \text{otherwise.} \end{cases}$
$\mathsf{let}\ v_1{=}x_1, \cdots, v_n{=}x_n \mid e$	$E(\mathsf{let}\ v_1{=}x_1 \mid \mathsf{let}\ v_2{=}x_2, \cdots, v_n{=}x_n \mid e)\mathfrak{i}$
α => e_1 else e_2	$\begin{cases} E(e_1)\mathfrak{i} & \text{if } \mathfrak{i} \models \alpha \\ E(e_2)\mathfrak{i} & \text{otherwise} \end{cases}$
$\{v_1{:}e_1, \ldots, v_n{:}e_n \mid \alpha\}$	$\{\langle a_1, \cdots, a_n\rangle \mid \left(\mathfrak{i} \oplus \left(\bigcup_{1\le i\le n}(v_i \mapsto E(e_i)\mathfrak{i})\right)\right) \models \alpha\}$

The function $\mathfrak{cal}{:}NumExpr \to \mathbb{Z}$ calculates the value of Alloy numeric expressions using 2's complement arithmetic. Its definition is given in the Table 3.2.

Notice that the multiplicity keywords in the arrow expression (m_1 and m_2 in $e_1 m_1$->$m_2 e_2$) are ignored here, but they will be used when such expressions participate in field and quantified variable declarations, to be defined later.

Tab. 3.1: Meaning of Alloy expressions.

n	$\mathfrak{cal}(n)\mathfrak{i}$
$\#e$	$\#E(e)\mathfrak{i}$
For numeric constant or variable n:	
n	$intVal(x)$ for the only x s.t. $\langle x\rangle \in \mathfrak{i}(n)$
e_1 add e_2	$\begin{cases} n_1+n_2 & \text{if } min_{\mathfrak{i}} \leq n_1+n_2 \leq max_{\mathfrak{i}} \\ min_{\mathfrak{i}}+n_1+n_2-max_{\mathfrak{i}} & \text{if } n_1+n_2 > max_{\mathfrak{i}} \\ max_{\mathfrak{i}}+n_1+n_2-min_{\mathfrak{i}} & \text{if } n_1+n_2 < min_{\mathfrak{i}} \end{cases}$
e_1 sub e_2	$\begin{cases} n_1-n_2 & \text{if } min_{\mathfrak{i}} \leq n_1-n_2 \leq max_{\mathfrak{i}} \\ min_{\mathfrak{i}}+(n_1-n_2)-max_{\mathfrak{i}} & \text{if } n_1-n_2 > max_{\mathfrak{i}} \\ max_{\mathfrak{i}}+(n_1-n_2)-min_{\mathfrak{i}} & \text{if } n_1-n_2 < min_{\mathfrak{i}} \end{cases}$
e_1 mul e_2	$\left[\ n_1n_2 \ // \ min_{\mathfrak{i}} + \Big(n_1n_2 - (max_{\mathfrak{i}}+1)\Big) \ \% \ \#\mathbb{Z}_{\mathfrak{i}} \Big) \ \right]$
e_1 div e_2	$\begin{cases} \left[\ \lfloor n_1/n_2 \rfloor \ // \ min_{\mathfrak{i}} \ \right] & \text{if } n_1 > 0 \text{ and } n_2 > 0 \\ -1 & \text{if } n_1 > 0 \text{ and } n_2 = 0 \\ \left[\ \lceil n_1/n_2 \rceil \ // \ max_{\mathfrak{i}} \ \right] & \text{if } n_1 > 0 \text{ and } n_2 < 0 \\ 0 & \text{if } n_1 = 0 \\ \left[\ \lceil n_1/n_2 \rceil \ // \ max_{\mathfrak{i}} \ \right] & \text{if } n_1 < 0 \text{ and } n_2 < 0 \\ \left[\ 1 \ // \ -1 \ \right] & \text{if } n_1 < 0 \text{ and } n_2 = 0 \\ \left[\ \lfloor n_1/n_2 \rfloor \ // \ min_{\mathfrak{i}} \ \right] & \text{if } n_1 < 0 \text{ and } n_2 > 0 \end{cases}$
e_1 rem e_2	$\begin{cases} 0 & \text{if } n_1 = 0 \\ n_1 & \text{if } n_1 \neq 0 \text{ and } n_2 = 0 \\ n_1 \ \% \ n_2 & \text{otherwise} \end{cases}$
e.sum int$[e]$	$\sum_{v\in E(e)\mathfrak{i}} intVal(v) \quad when\ \#E(e)\mathfrak{i} = 1$
sum $x_1{:}e_1, \cdots, x_z{:}e_z \mid e$	$\sum_{\substack{v\in E(e)\mathfrak{i}' \\ \mathfrak{i}'\in\{\mathfrak{i}\oplus\{x_i\mapsto a_i\}_{i=1}^{z} \mid a_i\in E(e_i)\mathfrak{i},\ 1\leq i\leq z\}}} intVal(v)$ $\begin{array}{l} when\ \#E(e_i)\mathfrak{i} = 1, \forall i, 1 \leq i \leq z \\ and\ \#E(e)\mathfrak{i}' = 1, for\ each\ \mathfrak{i}' \end{array}$
let $v = e \mid n$	$\mathfrak{cal}(n)\big(\mathfrak{i} \oplus \{v \mapsto E(e)\mathfrak{i}\}\big)$
let $v = n_1 \mid n$	$\mathfrak{cal}(n)\big(\mathfrak{i} \oplus \{v \mapsto \{\langle intVal^{-1}(\mathfrak{cal}(n_1)\mathfrak{i})\rangle\}\}\big)$
let $v = \alpha \mid n$	$\begin{cases} \mathfrak{cal}(n)\big(\mathfrak{i} \oplus \{v \mapsto \emptyset\}\big) & \text{if } \mathfrak{i} \models \alpha. \\ \mathfrak{cal}(n)\mathfrak{i} & \text{otherwise.} \end{cases}$
let $v_1{=}x_1, \cdots, v_z{=}x_z \mid n$	$\mathfrak{cal}(\text{let } v_1{=}x_1 \mid \text{let } v_2{=}x_2, \cdots, v_z{=}x_z \mid n)\mathfrak{i}$

In this definition, each n_j stands for the result of $\mathfrak{cal}(e_j)\mathfrak{i}$, where e_j is a numeric expression; $min_{\mathfrak{i}}(max_{\mathfrak{i}})$ denotes the minimum (maximum) integer defined in $\mathfrak{i}$; $\#\mathbb{Z}_{\mathfrak{i}}$ denotes the number of integer values defined in $\mathfrak{i}$ (note that $\#\mathbb{Z}_{\mathfrak{i}} = 2^{bw}$, where *bw* is the so called *bit width* of the instance). Additionally, the expression $\left[\ a \ // \ b \ \right]$ denotes a if $min_{\mathfrak{i}} \leq a \leq max_{\mathfrak{i}}$ and b in other case.

Tab. 3.2: Definition of the function $\mathfrak{cal}{:}NumExpr \rightarrow \mathbb{Z}$.

```
⟨formula⟩ ::= ⟨name⟩
   | let ⟨let-binding⟩,+ [| ⟨formula⟩ | ⟨conjunction⟩]
   | ⟨quant⟩ ⟨bindings⟩ [| ⟨formula⟩ | ⟨conjunction-block⟩]
   | [! | not] ⟨formula⟩
   | [ no | some | lone | one ] ⟨expression⟩
   | ⟨formula⟩ ⟨boolean-op⟩ ⟨formula⟩
   | ⟨expression⟩ [!|not]? [= | in] ⟨expression⟩
   | ⟨numeric-expr⟩ [!|not]? [= | < | > | =< | >=] ⟨numeric-expr⟩
   | ⟨formula⟩ [=>|implies] ⟨formula⟩ else ⟨formula⟩
   | ⟨pred⟩ [ ⟨expression⟩,*]
   | (⟨formula⟩)
   | ⟨conj-block⟩

⟨boolean-op⟩ ::= [ [|| | or] | [&& | and] | [<=> | iff] | [=> | implies] ]

⟨quant⟩ ::= all | no | some | lone | one

⟨conj-block⟩ ::= {⟨formula⟩*}

⟨bindings⟩ ::= ([disj] ⟨var⟩,+ : ⟨declaration-expr⟩),+

⟨declaration-expr⟩ ::= ⟨set-decl-expr⟩ | ⟨arrow-decl-expr⟩

⟨set-decl-expr⟩ ::= ⟨mult⟩? ⟨expression⟩

⟨arrow-decl-expr⟩ ::= ⟨arrow-decl-expr⟩ [⟨mult⟩]? -> [⟨mult⟩]? ⟨arrow-decl-expr⟩
   | ⟨expression⟩ [⟨mult⟩]? -> [⟨mult⟩]? ⟨arrow-decl-expr⟩
   | ⟨arrow-decl-expr⟩ [⟨mult⟩]? -> [⟨mult⟩]? ⟨expression⟩
   | ⟨expression⟩ [⟨mult⟩]? -> [⟨mult⟩]? ⟨expression⟩

⟨mult⟩ ::= some | one | lone |set

⟨pred⟩ ::= ⟨identifier⟩
```

Grammar diagram 3.2: Alloy grammar for formulas

The formulas in Alloy are mainly constructed from equalities and inclusions (in operator) and combining formulas using usual boolean operators (conjunction, disjunction, etc.) and quantifications. It is worth noting that the Alloy language admits higher-order quantifications, although the Alloy Analyzer does not perform analysis on specifications containing such kind of formulas. The Alloy language also provides useful syntax constructs, such as *if-then-else* formulas, *let* formulas, user-defined predicates, and conjunction block formulas.

In order to improve the readability of the definition of the meaning of Alloy formulas, we split it according to the complexity of the formula being analyzed.

Definition 3.1.6: (Satisfiability of Alloy formulas) We say that an Alloy formula α in the context of a specification S is satisfiable w.r.t. an instance $\mathfrak{i}$ (noted as $\mathsf{i} \models \alpha$) according to the following statements:

$\mathsf{i} \models \mathsf{no}\ e$	*iff*	$\mathfrak{i}(e) = \emptyset$
$\mathsf{i} \models \mathsf{some}\ e$	*iff*	$\mathfrak{i}(e) \neq \emptyset$
$\mathsf{i} \models \mathsf{lone}\ e$	*iff*	$\#\mathfrak{i}(e) \leq 1$
$\mathsf{i} \models \mathsf{one}\ e$	*iff*	$\#\mathfrak{i}(e) = 1$
$\mathsf{i} \models e_1 = e_2$	*iff*	$E(e_1)\mathfrak{i} = E(e_2)\mathfrak{i}$
$\mathsf{i} \models e_1\ [!\vert\mathsf{not}] = e_2$	*iff*	$E(e_1)\mathfrak{i} \neq E(e_2)\mathfrak{i}$
$\mathsf{i} \models e_1\ \mathsf{in}\ e_2$	*iff*	$E(e_1)\mathfrak{i} \subseteq E(e_2)\mathfrak{i}$
$\mathsf{i} \models e_1\ [!\vert\mathsf{not}]\ \mathsf{in}\ e_2$	*iff*	$E(e_1)\mathfrak{i} \nsubseteq E(e_2)\mathfrak{i}$
$\mathsf{i} \models n_1 = n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} = \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1\ [!\vert\mathsf{not}] = n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \neq \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1 < n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} < \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1\ [!\vert\mathsf{not}] < n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \not< \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1 <= n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \leq \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1\ [!\vert\mathsf{not}] = n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \nleq \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1 > n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} > \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1\ [!\vert\mathsf{not}] > n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \not> \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1 >= n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \geq \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models n_1\ [!\vert\mathsf{not}] >= n_2$	*iff*	$\mathfrak{cal}(n_1)\mathfrak{i} \ngeq \mathfrak{cal}(n_2)\mathfrak{i}$
$\mathsf{i} \models !\alpha$ $\mathsf{i} \models \mathsf{not}\ \alpha$	*iff*	not $\mathsf{i} \models \alpha$
$\mathsf{i} \models \alpha \mathbin{\vert\vert} \beta$ $\mathsf{i} \models \alpha\ \mathsf{or}\ \beta$	*iff*	$\mathsf{i} \models \alpha$ or $\mathsf{i} \models \beta$
$\mathsf{i} \models \alpha\ \&\&\ \beta$ $\mathsf{i} \models \alpha\ \mathsf{and}\ \beta$	*iff*	$\mathsf{i} \models \alpha$ and $\mathsf{i} \models \beta$
$\mathsf{i} \models \alpha <=> \beta$ $\mathsf{i} \models \alpha\ \mathsf{iff}\ \beta$	*iff*	$\mathsf{i} \models \alpha$ if and only if $\mathsf{i} \models \beta$
$\mathsf{i} \models \alpha => \beta$ $\mathsf{i} \models \alpha\ \mathsf{implies}\ \beta$	*iff*	either $\mathsf{i} \models \beta$, or not $\mathsf{i} \models \alpha$
$\mathsf{i} \models \mathsf{let}\ v = e \mid \alpha$	*iff*	$(\mathsf{i} \oplus \{v \mapsto E(e)\mathfrak{i}\}) \models \alpha$
$\mathsf{i} \models \mathsf{let}\ v = \alpha \mid \alpha$	*iff*	$\begin{cases} \mathsf{i} \oplus \{v \mapsto \emptyset\} \models \alpha & \textit{when } \mathsf{i} \models \alpha \\ \mathsf{i} \models \alpha & \textit{otherwise} \end{cases}$
$\mathsf{i} \models \mathsf{let}\ v = n \mid \alpha$	*iff*	$(\mathsf{i} \oplus \{v \mapsto \{\langle intVal^{-1}(\mathfrak{cal}(n)\mathfrak{i})\rangle\}\}) \models \alpha$
$\mathsf{i} \models \mathsf{let}\ v_1{=}x_1, \cdots, v_z{=}x_z \mid \alpha$	*iff*	$\mathsf{i} \models \mathsf{let}\ v_1{=}x_1 \mid \mathsf{let}\ v_2{=}x_2, \cdots, v_z{=}x_z \mid \alpha$
$\mathsf{i} \models v$	*iff*	$v \in dom(\mathsf{i})$
$\mathsf{i} \models \alpha => \beta\ \mathsf{else}\ \delta$	*iff*	$\mathsf{i} \models \beta$ when $\mathsf{i} \models \alpha$, and $\mathsf{i} \models \delta$ when $\mathsf{i} \not\models \alpha$

$\mathsf{i} \models p[e_1, \cdots, e_n]$	*iff*	$\left(\mathsf{i} \oplus \bigcup_{i=1}^{k} \{x_i \mapsto e_i\}\right) \models \alpha$ where p is the predicate declared as: $\mathsf{pred}\ p\ [x_1{:}e_1, \cdots, x_k{:}e_k]\{\alpha\}$
$\mathsf{i} \models \{\alpha_1 \cdots \alpha_n\}$	*iff*	$\mathsf{i} \models \alpha_1$ and, when $n > 1$, $\mathsf{i} \models \{\alpha_2 \cdots \alpha_n\}$
$\mathsf{i} \models \mathsf{all}\ v : d \mid \alpha$	*iff*	for all relation $R \subseteq \prod_1^{\mathfrak{a}(d)} E(\mathsf{univ})\mathsf{i}$, $\mathsf{i} \oplus \{v \mapsto R\} \models (v\ \mathsf{in} d'\ \&\&\ \mathcal{R}(v,d)) \Rightarrow \alpha$
$\mathsf{i} \models \mathsf{some}\ v : d \mid \alpha$	*iff*	exist a relation $R \subseteq \prod_1^{\mathfrak{a}(d)} E(\mathsf{univ})\mathsf{i}$ such that $\mathsf{i} \oplus \{v \mapsto R\} \models v\ \mathsf{in} d'\ \&\&\ \mathcal{R}(v,d)\ \&\&\ \alpha$
$\mathsf{i} \models \mathsf{no}\ v : d \mid \alpha$	*iff*	do not exist a relation $R \subseteq \prod_1^{\mathfrak{a}(d)} E(\mathsf{univ})\mathsf{i}$ s.t. $\mathsf{i} \oplus \{v \mapsto R\} \models v\ \mathsf{in} d'\ \&\&\ \mathcal{R}(v,d)\ \&\&\ \alpha$
$\mathsf{i} \models \mathsf{lone}\ v : d \mid \alpha$	*iff*	exist at most a relation $R \subseteq \prod_1^{\mathfrak{a}(d)} E(\mathsf{univ})\mathsf{i}$ s.t. $\mathsf{i} \oplus \{v \mapsto R\} \models v\ \mathsf{in} d'\ \&\&\ \mathcal{R}(v,d)\ \&\&\ \alpha$
$\mathsf{i} \models \mathsf{one}\ v : d \mid \alpha$	*iff*	exist exactly one relation $R \subseteq \prod_1^{\mathfrak{a}(d)} E(\mathsf{univ})\mathsf{i}$ s.t. $\mathsf{i} \oplus \{v \mapsto R\} \models v\ \mathsf{in} d'\ \&\&\ \mathcal{R}(v,d)\ \&\&\ \alpha$
$\mathsf{i} \models quant\ n_1, \cdots, n_m : d \mid f$	*iff*	$\mathsf{i} \models quant\ n_1 : d \mid (\mathsf{all}\ n_2, \cdots, n_m : d \mid f)$ where *quant* is all, some, no, lone or one.
$\mathsf{i} \models quant\ bind_1, \cdots, bind_n \mid f$	*iff*	$\mathsf{i} \models quant\ bind_1 \mid (\mathsf{all}\ bind_2, \cdots, bind_n \mid f)$ where *quant* is all, some, no, lone or one.

Where d' is the result of removing multiplicity keywords from d if any, and $\mathcal{R}(v,d)$ is the representation of the restriction stated by the declaration written in Alloy in the following way:

d	$\mathcal{R}(v,d)$
one e	one v
some e	some v
lone e	lone v
set e	*(v does not need further restrictions)*
For $n = \mathfrak{a}(d)$, $n_1 = \mathfrak{a}(d_1^A)$, $n_2 = \mathfrak{a}(d_2^A)$	
$d_1^A\ m_1 \texttt{->} m_2\ d_2^A$	$\begin{array}{l} \mathsf{all}\ v_1, \cdots, v_n : \mathsf{univ} \mid \\ \quad \mathsf{let}\ v_{left} = v.v_n.\cdots.v_{n_2},\ v_{right} = v_{n_1}.(\cdots.(v_1.v)\cdots) \mid \\ \qquad m_1\ v_{left}\ \&\&\ m_2\ v_{right}\ \&\&\ \mathcal{R}(v_{left}, d_1^A)\ \&\&\ \mathcal{R}(v_{right}, d_2^A) \end{array}$
Being $n = \mathfrak{a}(d)$, $n_1 = \mathfrak{a}(e_1)$, $n_2 = \mathfrak{a}(d_2^A)$	
$e_1\ m_1 \texttt{->} m_2\ d_2^A$	$\begin{array}{l} \mathsf{all}\ v_1, \cdots, v_n : \mathsf{univ} \mid \\ \quad \mathsf{let}\ v_{left} = v.v_n.\cdots.v_{n_2},\ v_{right} = v_{n_1}.(\cdots.(v_1.v)\cdots) \mid \\ \qquad m_1\ v_{left}\ \&\&\ m_2\ v_{right}\ \&\&\ \mathcal{R}(v_{right}, d_2^A) \end{array}$
Being $n = \mathfrak{a}(d)$, $n_1 = \mathfrak{a}(d_1^A)$, $n_2 = \mathfrak{a}(e_2)$	
$d_1^A\ m_1 \texttt{->} m_2\ e_2$	$\begin{array}{l} \mathsf{all}\ v_1, \cdots, v_n : \mathsf{univ} \mid \\ \quad \mathsf{let}\ v_{left} = v.v_n.\cdots.v_{n_2},\ v_{right} = v_{n_1}.(\cdots.(v_1.v)\cdots) \mid \\ \qquad m_1\ v_{left}\ \&\&\ m_2\ v_{right}\ \&\&\ \mathcal{R}(v_{left}, d_1^A) \end{array}$
Being $n = \mathfrak{a}(d)$, $n_1 = \mathfrak{a}(e_1)$, $n_2 = \mathfrak{a}(e_2)$	
$e_1\ m_1 \texttt{->} m_2\ e_2$	$\begin{array}{l} \mathsf{all}\ v_1, \cdots, v_n : \mathsf{univ} \mid \\ \quad \mathsf{let}\ v_{left} = v.v_n.\cdots.v_{n_2},\ v_{right} = v_{n_1}.(\cdots.(v_1.v)\cdots) \mid \\ \qquad m_1\ v_{left}\ \&\&\ m_2\ v_{right} \end{array}$

Now we can define the notion of semantic consequence of Alloy formulas.

$\langle$*signature decl*$\rangle$ ::= $\langle$*primitive sig-decl*$\rangle$ | $\langle$*subset sig-decl*$\rangle$

$\langle$*primitive sig-decl*$\rangle$ ::= [$\langle$*sig.qual.*$\rangle$] `sig` $\langle$*sig*$\rangle_{,+}$ [`extends` $\langle$*sig*$\rangle$]
`{`$\langle$*fields*$\rangle$`}` [$\langle$*conj-block*$\rangle$]

$\langle$*subset sig-decl*$\rangle$ ::= [$\langle$*sig.qual.*$\rangle$] `sig` $\langle$*sig*$\rangle_{,+}$ `in` $\langle$*name*$\rangle$ [`+` $\langle$*sig*$\rangle$]*
`{`$\langle$*fields*$\rangle$`}` [$\langle$*conj-block*$\rangle$]

$\langle$*sig.qual.*$\rangle$ ::= `abstract` | `lone` | `one` | `some`

$\langle$*fields*$\rangle$ ::= [$\langle$*field*$\rangle_{,+}$`:` $\langle$*declaration-expr*$\rangle$]$_{,*}$]

Grammar diagram 3.3: Grammar of signature declarations.

Definition 3.1.7: (Semantic consequence) Let α an Alloy formula and Σ a set of Alloy formulas. We will say that α *is a semantic consequence of* Σ (denoted by $\Sigma \models \alpha$) if for each instance $\mathfrak{a}$ such that $\mathfrak{a} \models \sigma$ for all $\sigma \in \Sigma$, it also holds that $\mathfrak{a} \models \alpha$.

Declarations and implicit facts

The declarations in the specification state which symbols belong to the domain of every legal instance and also determine facts that must be fulfilled by the image of those symbols. In this section, we will explore how each declaration affects the constraints imposed on an instance to be called legal. Diagram 3.3 shows the grammar of the signature declarations.

For every signature definition **sig** S { } a legal instance $\mathfrak{i}$ must meet that:

$$\mathsf{S} \in dom(\mathfrak{i}) \quad \text{and} \quad \mathsf{Arity}(\mathfrak{i}(\mathsf{S})) = 1 \tag{3.1}$$

Alloy allows the definition of two kinds of signatures: *primitive signatures* and *subset signatures*. The primitive signatures can be arranged forming an inheritance hierarchy by using the keyword extends. Every legal instance must obey this hierarchy, which is achieved by assuring that, for each primitive signature S,

$$\bigcup_{\substack{\mathsf{T}\ \text{s.t.}\\ \mathsf{sig}\ \mathsf{T}\ \mathsf{extends}\ \mathsf{S}}} \mathfrak{i}(\mathsf{T}) \subseteq \mathfrak{i}(\mathsf{S}) \tag{3.2}$$

Additionally, if the definition of S bears the modifier abstract, for the instance $\mathfrak{i}$ to be considered legal the inverse inclusion must hold too.

$$\bigcup_{\substack{\mathsf{T}\ \text{s.t.}\\ \mathsf{sig}\ \mathsf{T}\ \mathsf{extends}\ \mathsf{S}}} \mathfrak{i}(\mathsf{T}) \supseteq \mathfrak{i}(\mathsf{S}) \quad \text{when S is declared abstract} \tag{3.3}$$

The subset signatures can be seen as a way to embrace atoms from one or more other signatures. A legal instance $\mathfrak{i}$ must then fulfill that, given a definition such as **sig** S $\{\cdots\}$**in** T_1+$\cdots$+T_n

$$\mathfrak{i}(\mathsf{S}) \subseteq \bigcup_{i=1}^{n} \mathfrak{i}(\mathsf{T}_i) \tag{3.4}$$

The subset signatures can not be abstract.

The multiplicity qualifiers lone, one and some affects the cardinality of the image of the signature symbol on $\mathfrak{i}$ according to the following statements.

$$\begin{array}{lll} \#\mathfrak{i}(\mathsf{S}) = 1 & \text{if} & \mathsf{S} \text{ is defined as } \mathsf{one\ sig} \\ \#\mathfrak{i}(\mathsf{S}) \leq 1 & \text{if} & \mathsf{S} \text{ is defined as } \mathsf{lone\ sig} \\ \#\mathfrak{i}(\mathsf{S}) \geq 1 & \text{if} & \mathsf{S} \text{ is defined as } \mathsf{some\ sig} \end{array} \tag{3.5}$$

Similarly to signatures, each field symbol must belong to the domain of the legal instances and the relation mapped to it has to match the arity of the declaring expression. A declaration such as sig $\mathsf{S}\{\cdots \mathsf{f} : \mathsf{d} \cdots\}$ establish the following restrictions on $\mathfrak{i}$:

$$\mathsf{f} \in dom(\mathfrak{i}) \quad \text{and} \quad \mathfrak{i}(\mathsf{f}) \subseteq \prod_{1}^{\mathfrak{a}(d)} E(\mathsf{univ})\mathfrak{i} \tag{3.6}$$

Additionally, it also imposes a restriction on $\mathfrak{i}(\mathsf{f})$ similar to those imposed by the declaring expression of a quantification.

$$\begin{array}{rl} \mathfrak{i} \models \mathcal{R}_{\mathfrak{i}}(\mathsf{S}- > \mathsf{one}\ d)\mathsf{f} & \text{if } \mathfrak{a}(d) = 1 \text{ and has no multiplicity keyword} \\ \mathfrak{i} \models \mathcal{R}_{\mathfrak{i}}(\mathsf{S}- > d)\mathsf{f} & \text{otherwise} \end{array} \tag{3.7}$$

Finally, the conjunction block placed after the signature declaration states restrictions on the elements denoted by the signature and fields declared there. Given the declaration sig $\mathsf{S}\{\mathsf{f}_1{:}d_1,\ \cdots,\ \mathsf{f}_m{:}d_m\}\{\alpha_1 \cdots \alpha_n\}$ a legal instance for it must satisfy:

$$\mathfrak{i} \models \mathfrak{r}(\alpha_i) \quad \text{for every } i \text{ s.t. } 1 \leq i \leq n \tag{3.8}$$

where $\mathfrak{r}$ is a function that, given an Alloy formula, returns another Alloy formula that only differs from the argument in that every symbol $\mathsf{f_j}$ not preceded by an @ is replaced by $\mathsf{S.f_j}$; when preceded by @, $\mathfrak{r}$ just removes it.

Paragraphs and Modules

Every Alloy specification is constructed as a hierarchy of Alloy *modules*. Each of them is intended to encapsulate a conceptual piece of the whole specification. A module may contain *signature*, *fact*, *function*, *predicate*, *assertion*, and *run* declarations, a shown in diagram 3.4.

$\langle$*specification*$\rangle$ ::= [`module` $\langle$*module name*$\rangle$] $\langle$*import declaration*$\rangle$* $\langle$*paragraph*$\rangle$*

$\langle$*import declaration*$\rangle$ ::= `open` $\langle$*module name*$\rangle$

$\langle$*paragraph*$\rangle$ ::= $\langle$*signature declaration*$\rangle$
| $\langle$*fact declaration*$\rangle$
| $\langle$*assertion declation*$\rangle$
| $\langle$*function declaration*$\rangle$
| $\langle$*predicate declaration*$\rangle$
| $\langle$*command declaration*$\rangle$

$\langle$*command declaration*$\rangle$ ::= $\langle$*check declaration*$\rangle$ | $\langle$*run declaration*$\rangle$

Grammar diagram 3.4: Grammar of Alloy specifications and modules.

The signature declarations were extensively discussed in the previous section. The predicate and function declarations are the way to include parametrized named schemes of formulas and expressions, in order to facilitate the reuse of concepts.

As mentioned, the *fact declarations* impose constraints that must be fulfilled by every legal instance of the specification. On the other hand, the *assertion declarations* state hypotheses about the universe modeled by the specification. These hypotheses can be checked using the Alloy Analyzer. To do that, a *check command declaration* must be included in the specification. The Alloy Analyzer also can be used to check the consistency of a predicate. To do so, the specification must include a *run command declaration.*

The hierarchy of modules is constructed using the keyword open. The modules are able to receive parameters. Then, when using another module, the arguments must be given in the *import declaration.* As this is not a key aspect for the present work, the interested reader is pointed to Jackson (2012), where all the details on the matter can be found.

3.2 A Complete Calculus Useful for Alloy Users

In this section we will present a deductive calculus useful for the verification of Alloy assertions. The procedure we will follow in order to present the calculus is the following:

- We will present the class of "proper point-dense omega closure fork algebras". These algebras contain operations akin to Alloy operations. We will also present a complete calculus for this class of algebras. The deduction relation in this formalism will be denoted by $\vdash_{\text{FORK}}$.

- We will present an interpretability theorem from Alloy theories to fork algebra theories. An interpretability theorem, in this context, consists of a mapping $\mathbf{F} : AlloyForm \to ForkForm$ (mapping Alloy formulas to fork formulas), and a theorem proving that:

$$\Gamma \models_{Alloy} \alpha \quad \iff \quad \{\, F(\gamma) : \gamma \in \Gamma \,\} \vdash_{\textsc{Fork}} F(\alpha) \ .$$

Notice that checking the validity of an Alloy assertion in a specification reduces to the problem of proving a property in the deductive calculus of fork algebras. Since the fork-algebraic formalism is not exactly Alloy, it is essential to discuss to what extent is the new formalism useful for Alloy users. This discussion permeates Sections 3.2.1–3.2.3.

In Section 3.2.1 we present the fork formalism. In Section 3.2.2 we discuss how Alloy quantification is modeled in a formalism where quantifiers range over relations. In Section 3.2.3, we present the interpretability result. Notice that a particular theory that has to be interpreted in the algebraic formalism is the Alloy theory for integers (c.f. Section 3.2.2).

3.2.1 Proper Point-Dense Omega Closure Fork Algebras

We begin this section by introducing the class of *proper* point-dense omega closure fork algebras. Qualifier "proper" refers to the fact that these algebras are special in the sense that they are particularly close to the semantics of Alloy. In effect, these algebras have (binary) relations (on a given set B) in their universe, and operations for union, intersection, difference[1], navigation, transposition and closure of relations, as Alloy has.

Definition 3.2.1: (proper PDOCFA) A *proper* point-dense omega closure fork algebra on a set B is a structure

$$\langle\, \mathbf{R}, +, \&, ^{-}, \emptyset, univ, ., iden, \sim, *, \nabla \,\rangle$$

where:

$\mathbf{R}$ is a set of binary relations on the set B, closed under the operations.

$^{-}$ is set-complement, $+$ is set-union and & is set-intersection.

$\emptyset$ is the empty set, and *univ* is the binary relation $B \times B$.

$.$ is composition between binary relations.

iden is the identity relation on B.

$\sim$ is transposition of binary relations.

$*$ is reflexive-transitive closure of binary relations.

[1] Actually, these algebras have a complement operation, but the latter allows us to define difference with the aid of intersection.

∇ is the *fork* operation. It is defined as

$$S\nabla T = \{\, \langle a, b \star c\rangle : \langle a,b\rangle \in S \;\wedge\; \langle a,c\rangle \in T \,\} \;. \qquad (3.9)$$

Symbol $\star$ in (3.9) stands for an injective function of type $B \times B \to B$. Therefore, we assume set B to be closed under $\star$.

Notation 1: We will often name proper-PDOCFA structures using capital gothic letters (such as $\mathfrak{A}$) and the set of its binary relations using a bold capital R with the proper-PDOCFA it belongs as a subscript (for example: $\mathbf{R}_{\mathfrak{A}}$).

Syntax and Semantics of the Language

Before going further, we formally present the languages we propose as the target of the translation of Alloy specifications. As usual when working with logical languages, we define a family of languages called $\mathscr{L}$-PDOCFA. Their main features are the following.

- A countable infinite collection of variables.
- A finite collection of constants.
- Symbols for the constants and operations from def. 3.2.1 (none, univ, iden, ~, $*$, , +, &, ., ∇).
- boolean connectives for negation (!), disjunction (||), conjunction (&&), logical implication ($\Rightarrow$), and material equivalence ($\Leftrightarrow$).
- Universal and existential quantifiers (all and some).
- Syntax constructions resembling Alloy constructions, as *let expressions* and *if-then-else expressions*.

The constants and variables in these languages are pairs formed by an alpha-numeric word w and a natural number k. We write them as $w^{(k)}$. Although to use such pairs as constants and variables may seem unnecessary, their usefulness will be clear when the translation will be presented.

The grammar to construct expressions and formulas is shown in diagram 3.5. As the reader can see, there are several aspects shared by Alloy and the languages in $\mathscr{L}$-PDOCFA. We look for this similarity in order to simplify the translation process.

As with Alloy, we will use the notion of relational instance in order to assign meaning to expressions.

Definition 3.2.2: (Relational instance for a $\mathscr{L}$-PDOCFA language) Let L be a language from $\mathscr{L}$-PDOCFA, and $\mathfrak{A}$ a proper-PDOCFA. A *relational instance* for L on $\mathfrak{A}$ (denoted by $\mathfrak{p}^{L}_{\mathfrak{A}}$) is a mapping that associates every constant and variable symbol of L with a binary relation in $\mathbf{R}_{\mathfrak{A}}$.

$\langle expression \rangle$ ::= none | univ | iden | $\langle constant\text{-}symbol \rangle$
  | [$\sim$ | $*$] $\langle expression \rangle$ | $\langle expression \rangle$
  | $\langle expression \rangle$ [+ | & | . | ∇] $\langle expression \rangle$
  | `let` $\langle variable\text{-}symbol \rangle$ = [$\langle expression \rangle$ | $\langle formula \rangle$] | $\langle expression \rangle$
  | $\langle formula \rangle \Rightarrow \langle expression \rangle$ `else` $\langle expression \rangle$

$\langle constant\text{-}symbol \rangle$ ::= $\langle word \rangle^{(\langle number \rangle)}$

$\langle variable\text{-}symbol \rangle$ ::= $\langle word \rangle^{(\langle number \rangle)}$

$\langle word \rangle$ ::= [[A-Z][a-z]]$^+$[[A-Z][a-z][0-9]]*

$\langle number \rangle$ ::= [1-9]$^+$[0-9]*

$\langle formula \rangle$ ::= $\langle expression \rangle$ = $\langle expression \rangle$
  | ! $\langle formula \rangle$
  | $\langle formula \rangle$ [|| | && | $\Rightarrow$ | $\Leftrightarrow$] $\langle formula \rangle$
  | `let` $\langle variable\text{-}symbol \rangle$ = [$\langle expression \rangle$ | $\langle formula \rangle$] | $\langle formula \rangle$
  | $\langle formula \rangle \Rightarrow \langle formula \rangle$ `else` $\langle formula \rangle$
  | `all` $\langle variable\text{-}symbol \rangle$ | $\langle formula \rangle$ | `some` $\langle variable\text{-}symbol \rangle$ | $\langle formula \rangle$

Grammar diagram 3.5: Basic PDOCFA grammar

The collection of all possible relational instances for a language L on a proper-PDOCFA $\mathfrak{A}$ will be denoted by $\mathscr{I}_{L,\mathfrak{A}}$. For the sake of clarity, the super and subscripts in $\mathfrak{p}^L_{\mathfrak{A}}$ and $\mathscr{I}_{L,\mathfrak{A}}$ will be used only when necessary.

We will refer to $\mathscr{E}^{\text{PDOCFA}}_L$ as the collection of all expressions that can be generated by a $\mathscr{L}$-PDOCFA L. We will define a meta-function that, given an expression $e \in \mathscr{E}^{\text{PDOCFA}}_L$, for a $\mathscr{L}$-PDOCFA L, and a relational instance $\mathfrak{p}^L_{\mathfrak{A}}$, calculates the relation denoted by e in the context of the instance $\mathfrak{p}^L_{\mathfrak{A}}$. This meta function can be seen as the homomorphic extension of the relational instance $\mathfrak{p}^L_{\mathfrak{A}}$.

Definition 3.2.3: (Meaning of $\mathscr{L}$-PDOCFA expressions) Let L be a language from $\mathscr{L}$-PDOCFA, $\mathfrak{A}$ a proper-PDOCFA, and $\mathbf{R}_{\mathfrak{A}}$ the set of binary relations from $\mathfrak{A}$. The meta-function

$$\mathcal{X}^L_{\mathfrak{A}} : \mathscr{E}^{\text{PDOCFA}}_L \to \mathscr{I}_{L,\mathfrak{A}} \to \mathbf{R}_{\mathfrak{A}}$$

whose detailed definition is shown in table 3.3, assigns meaning to $\mathscr{L}$-PDOCFA expressions based on a particular instance.
Symbolically, if $e \in \mathscr{E}^{\text{PDOCFA}}_L$ and $\mathfrak{p}^L_{\mathfrak{A}} \in \mathscr{I}_{L,\mathfrak{A}}$, then $\mathcal{X}^L_{\mathfrak{A}}(e)\mathfrak{p}^L_{\mathfrak{A}}$ is the binary relation denoted by e in the context of $\mathfrak{p}^L_{\mathfrak{A}}$.

The notion of semantic truth is defined similarly to the Alloy case.

e	$\mathcal{X}^L_{\mathfrak{A}}(e)\mathfrak{p}^L_{\mathfrak{A}}$
none	$\emptyset$
univ	$univ$
iden	$iden$
$c^{(k)}$	$\mathfrak{p}^L_{\mathfrak{A}}(c^{(k)})$
$v^{(k)}$	$\mathfrak{p}^L_{\mathfrak{A}}(v^{(k)})$
$\sim e$	$\sim \mathcal{X}^L_{\mathfrak{A}}(e)\mathfrak{p}^L_{\mathfrak{A}}$
$*e$	$*\mathcal{X}^L_{\mathfrak{A}}(e)\mathfrak{p}^L_{\mathfrak{A}}$
$\overline{e}$	$\overline{\mathcal{X}^L_{\mathfrak{A}}(e)\mathfrak{p}^L_{\mathfrak{A}}}$
$e_1 + e_2$	$\mathcal{X}^L_{\mathfrak{A}}(e_1)\mathfrak{p}^L_{\mathfrak{A}} + \mathcal{X}^L_{\mathfrak{A}}(e_2)\mathfrak{p}^L_{\mathfrak{A}}$
$e_1 \& e_2$	$\mathcal{X}^L_{\mathfrak{A}}(e_1)\mathfrak{p}^L_{\mathfrak{A}} \& \mathcal{X}^L_{\mathfrak{A}}(e_2)\mathfrak{p}^L_{\mathfrak{A}}$
$e_1 . e_2$	$\mathcal{X}^L_{\mathfrak{A}}(e_1)\mathfrak{p}^L_{\mathfrak{A}} . \mathcal{X}^L_{\mathfrak{A}}(e_2)\mathfrak{p}^L_{\mathfrak{A}}$
$e_1 \nabla e_2$	$\mathcal{X}^L_{\mathfrak{A}}(e_1)\mathfrak{p}^L_{\mathfrak{A}} \nabla \mathcal{X}^L_{\mathfrak{A}}(e_2)\mathfrak{p}^L_{\mathfrak{A}}$
let $v = e_1 \mid e_2$	$\mathcal{X}^L_{\mathfrak{A}}(e_2)\left(\mathfrak{p}^L_{\mathfrak{A}} \oplus \{v \mapsto e_1\}\right)$
let $v = \alpha \mid e$	$\begin{cases} \mathcal{X}^L_{\mathfrak{A}}(e)\left(\mathfrak{p}^L_{\mathfrak{A}} \oplus \{v \mapsto \emptyset\}\right) & \text{if } \alpha \text{ is true on } \mathfrak{p}^L_{\mathfrak{A}} \\ \mathcal{X}^L_{\mathfrak{A}}(e)\mathfrak{p}^L_{\mathfrak{A}} & \text{otherwise} \end{cases}$
$\alpha \Rightarrow e_1$ else e_2	$\begin{cases} \mathcal{X}^L_{\mathfrak{A}}(e_1)\mathfrak{p}^L_{\mathfrak{A}} & \text{if } \alpha \text{ is true on } \mathfrak{p}^L_{\mathfrak{A}} \\ \mathcal{X}^L_{\mathfrak{A}}(e_2)\mathfrak{p}^L_{\mathfrak{A}} & \text{otherwise} \end{cases}$

Note that the operation symbols in the right column represent the operations of Def. 3.2.1, with the exception of $\oplus$ (representing relational override) and $\mapsto$ (representing a maping).

Tab. 3.3: Meaning of $\mathscr{L}$-PDOCFA expressions.

DEFINITION 3.2.4: (Satisfiability of PDOCFA formulas) We will say that a PDOCFA formula α in the context of a language L, $L \in \mathscr{L}$-PDOCFA, is *satisfiable w.r.t. a proper*-PDOCFA $\mathfrak{A}$ *and an instance* $\mathfrak{p}^L_{\mathfrak{A}}$ (denoted by $\mathfrak{A}, \mathfrak{p}^L_{\mathfrak{A}} \models_{\mathsf{PDOCFA}} \alpha$) according to the following statements:

$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} e_1 = e_2$	*iff*	$\mathcal{X}_{\mathfrak{A}}^{L}(e_1)\mathfrak{p}_{\mathfrak{A}}^{L} = \mathcal{X}_{\mathfrak{A}}^{L}(e_2)\mathfrak{p}_{\mathfrak{A}}^{L}$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} !\alpha$	*iff*	not $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha \mid\mid \beta$	*iff*	$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha$ or $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \beta$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha \;\&\&\; \beta$	*iff*	$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha$ and $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \beta$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha \Rightarrow \beta$	*iff*	not $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha$ or $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \beta$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha \Leftrightarrow \beta$	*iff*	$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha$ if and only if $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \beta$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \mathsf{let}\ v{=}e \mid \alpha$	*iff*	$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \oplus \{v \mapsto \mathcal{X}_{\mathfrak{A}}^{L}(e)\mathfrak{p}_{\mathfrak{A}}^{L}\} \models_{\mathsf{PDOCFA}} \alpha$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \mathsf{let}\ v{=}\delta \mid \alpha$	*iff*	$\begin{cases} \mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \oplus \{v \mapsto \emptyset\} \models_{\mathsf{PDOCFA}} \alpha & \text{when } \mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \delta \\ \mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha & \text{otherwise} \end{cases}$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha \Rightarrow \beta\ \mathsf{else}\ \gamma$	*iff*	$\begin{cases} \mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \beta & \text{when } \mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha \\ \mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \gamma & \text{otherwise} \end{cases}$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \mathsf{all}\ v \mid \alpha$	*iff*	for all relation $R \in \mathbf{R}_{\mathfrak{A}}$, $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \oplus \{v \mapsto R\} \models_{\mathsf{PDOCFA}} \alpha$
$\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \mathsf{some}\ \ v \mid \alpha$	*iff*	exists some relation $R \in \mathbf{R}_{\mathfrak{A}}$, $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \oplus \{v \mapsto R\} \models_{\mathsf{PDOCFA}} \alpha$

We will say that a proper PDOCFA $\mathfrak{A}$ *satisfies* a PDOCFA formula α from the language L (denoted by $\mathfrak{A} \models \alpha$) if there is a relational instance $\mathfrak{p}_{\mathfrak{A}}^{L}$ for $\mathfrak{A}$ such that $\mathfrak{A}, \mathfrak{p}_{\mathfrak{A}}^{L} \models_{\mathsf{PDOCFA}} \alpha$.

3.2.2 Translating From Alloy to $\mathscr{L}$-PDOCFA

The first step of the translation process, starting from an Alloy specification S, will be to build a language $L \in \mathscr{L}$-PDOCFA, where every Alloy constant defined by the user in S has its PDOCFA counterpart in L. We will say that the language L and the Alloy specification S are *compatible* when their constants can be correlated in this way.

Although the languages in $\mathscr{L}$-PDOCFA share most of the operations with Alloy (at least intentionally and notationally), there are remarkable differences, such as: (a) while the relations defined in Alloy can have arbitrary arity, using PDOCFA only binary relations can be manipulated; (b) PDOCFA offers the fork operator, which is not directly tied to any Alloy operator; (c) PDOCFA bears the request for point-density; (d) at least apparently, PDOCFA lacks support for (bounded) integers; and (e) while quantification in Alloy ranges over atomic elements from signatures, in PDOCFA, quantifiers range over all the relations in the domain.

In the next sections we will show how these gaps can be bridged in order to provide Alloy with a complete calculus. Thus, the family of languages presented in this section serves as the core of the languages actually used by the translation, and it will be augmented in the next sections in order to support more suitably all the Alloy features.

```
sig Dir extends Object {
   contents:  Name −> lone Object,
   parent:  lone Dir
}
```

Fig. 3.1: Declaration of signature Dir.

Codifying Alloy relations in proper-PDOCFA

Since the way to define basic elements in Alloy is to declare signatures, we first turn to their translation. Notice that while function $\star$ in Definition 3.2.1 has to be injective, it need not be surjective. Therefore, there may exist elements in the base set B that do not encode pairs. These elements are called *urelements*. Every Alloy unary relation will be paired with a binary partial identity of urelements, i.e. a binary relation formed only by urelements, contained in the identity *iden*. For example, the translation of the signature Dir in Figure 3.1 will result in a PDOCFA constant $\mathsf{iden}^{(2)}_{\mathsf{Dir}}$.

Notice that translating Alloy unary relations in this way imposes a constraint on the proper-PDOCFAs that could be used as interpretations of the $\mathscr{L}$-PDOCFAs resulting from the translation of Alloy specifications. We will say that a proper-PDOCFA $\mathfrak{A}$ *can support* a language $L \in \mathscr{L}$-PDOCFA which is the outcome of the translation of a given Alloy specification S, if it contains a partial urelement identity for each signature, and these partial identities mimic the inheritance hierarchy of the Alloy signatures, as explained in Section 3.1 (Pg. 27).

To relate more complex Alloy terms with PDOCFA terms we must find an adequate means for modeling relations of arity greater than two as binary relations. The operator fork allows us to do this in a simple way. For each Alloy relation of arity at least binary holding tuples of the form $\langle a_1, a_2, \ldots, a_n\rangle$, the corresponding PDOCFA binary relation resulting from the translation process will hold tuples of the form[2] $\langle a_1, a_2 \star \cdots \star a_n\rangle$. Although the latter relation is always binary, we will say its rank is n if it is the result of the translation of an Alloy relation of arity n.

Notation 2: Being A an Alloy expression, and R its corresponding binary relation in a proper-PDOCFA, we will call $rank(R)$ the arity of A.

Let us illustrate this codification with an example. In section 2.1.1 we showed the definition depicted in Figure 3.1. Alloy field contents is a ternary relation. Then, its PDOCFA counterpart $\mathsf{contents}^{(3)}$ satisfies that given an alloy instance $\mathfrak{a}_S$, it is possible to find a relational instance $\mathfrak{p}^L_{\mathfrak{A}}$, where S and L are compatible, such that:

$$\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{contents}^{(3)}) = \{\, \langle a, b \star c\rangle : \langle a, b, c\rangle \in \mathfrak{a}_S(\mathsf{contents})\,\} \ .$$

[2] Since $\star$ is not associative, an expression of the form $a\star b\star c$ denotes the object $a\star(b\star c)$.

Furthermore, the domain restriction imposed on the field $\mathsf{contents}$ by its declaration, assuring that the tuples in $\mathsf{contents}$ are formed by atoms from signatures Dir, Name, and Object, in that order, can be stated on the corresponding PDOCFA constant in algebraic terms forcing that $\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{contents}^{(3)})$ be included in

$$\left(\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{iden}^{(1)}_{\mathsf{Dir}}).univ.\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{iden}^{(1)}_{\mathsf{Name}})\right) \nabla \left(\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{iden}^{(1)}_{\mathsf{Dir}}).univ.\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{iden}^{(1)}_{\mathsf{Object}})\right) .$$

Notice that the relations denoted by terms such as the one just depicted belongs to each PDOCFA structure $\mathfrak{A}$ that can support a language containing the translation of the signatures, such as $\mathsf{iden}^{(1)}_{\mathsf{Dir}}$, $\mathsf{iden}^{(1)}_{\mathsf{Name}}$, and $\mathsf{iden}^{(1)}_{\mathsf{Object}}$, since the set of relations $\mathbf{R}_{\mathfrak{A}}$ of $\mathfrak{A}$ is closed under the operations of the algebra.

Regarding individual Alloy variables declared in first order quantifications, our convention is that these are modeled using relational variables ranging over points. The Alloy language also allows the user to write higher order quantifications, although the Alloy Analyzer rarely is able to check them. Similarly to the translation of signatures and fields, the higher-order Alloy variables are mapped to PDOCFA variables that are not restricted by the constraint of being a point. Additionally, we use the number in the PDOCFA variables and constants in order to express the codified rank of the relation denoted by the corresponding symbol. In the previous example, the number 3 in $\mathsf{contents}^{(3)}$ indicates that this constant denotes a binary relation codifying a ternary relation.

Given an Alloy instance, we can characterize those proper PDOCFA that are candidates to interpret it. The way to formalize the correlation between Alloy constants and variables and their corresponding PDOCFA elements, is to show that it is possible to build a relational instance from an Alloy one, so that the correlation between symbols holds. The construction is done as follows.

Definition 3.2.5: Given an Alloy specification S and an instance $\mathfrak{a}_S$ for it, we define a PDOCFA language L and a relational instance $\mathfrak{p}^L_{\mathfrak{A}}$ for L such that L is the smallest language fulfilling $dom(\mathfrak{p}^L_{\mathfrak{A}}) \subseteq L$, and:

- If S is a signature,

$$\mathsf{iden}^{(1)}_{\mathsf{S}} \in L \quad \text{and} \quad \mathfrak{p}^L_{\mathfrak{A}}(\mathsf{iden}^{(1)}_{\mathsf{S}}) = \{\, \langle s, s\rangle : s \in \mathfrak{a}_S(\mathsf{S}) \,\} .$$

- If F is an n-ary field (recall that $n \geq 2$ is the only possible case) then $\mathsf{F}^{(n)} \in L$ and

$$\mathfrak{p}^L_{\mathfrak{A}}(\mathsf{F}^{(n)}) = \{\, \langle a_1, a_2 \star \cdots \star a_n\rangle : \langle a_1, a_2, \ldots, a_n\rangle \in \mathfrak{a}_S(\mathsf{F}) \,\} .$$

- If v is an Alloy variable, then

– if v range over Alloy atoms,

$$\mathsf{v}^{(1)} \in L \quad \text{and} \quad \mathfrak{p}^{L}_{\mathfrak{A}}(\mathsf{v}^{(1)}) = \{ \langle \mathfrak{a}_S(\mathsf{v}), \mathfrak{a}_S(\mathsf{v}) \rangle \} .$$

(Notice that the resulting relation is indeed a point.)

– else, calling n the arity of the Alloy relation denoted by v, $\mathsf{v}^{(n)} \in L$ and

$$\mathfrak{p}^{L}_{\mathfrak{A}}(\mathsf{v}^{(n)}) = \{ \langle a_1, a_2 \star \cdots \star a_n \rangle : \langle a_1, a_2, \ldots, a_n \rangle \in \mathfrak{a}_S(\mathsf{v}) \} .$$

(The resulting relation $\mathfrak{p}^{L}_{\mathfrak{A}}(\mathsf{v}^{(n)})$ will codify the arity of $\mathfrak{a}_S(\mathsf{v})$.)

Similarly, given a proper PDOCFA and a relational instance we can define a sort of canonical Alloy instance.

Definition 3.2.6: Let $L \in \mathscr{L}$-PDOCFA be an outcome of a translation from an Alloy specification, $\mathfrak{A}$ a proper PDOCFA that can support L, and $\mathfrak{p}^{L}_{\mathfrak{A}}$ a relational instance assigning meaning to constant and variable symbols in L. We construct an Alloy specification S and an Alloy instance $\mathfrak{a}_S$ as follows.

- For every partial identity symbol $\mathsf{iden}^{(1)}_{s} \in L$, there is a signature s in S such that:
$$\mathfrak{a}_S(s) = \left\{ a : \langle a, a \rangle \in \mathfrak{p}^{L}_{\mathfrak{A}}(\mathsf{iden}^{(1)}_{s}) \right\} ,$$
- For each constant symbol $c^{(k)} \in L$, not being a logical constant (such as: univ, iden or none) there is a field c in S such that:
$$\mathfrak{a}_S(c) = \left\{ \langle a_1, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in \mathfrak{p}^{L}_{\mathfrak{A}}(c^{(k)}) \right\} .$$
- For every variable symbol $v^{(k)} \in dom(\mathfrak{p}^{L}_{\mathfrak{A}})$,
$$\mathfrak{a}_S(v) = \begin{cases} \{ \langle a \rangle : \langle a, a \rangle \in \mathfrak{p}^{L}_{\mathfrak{A}}(v^{(k)}) \} & \text{if } k = 1, \\ \{ \langle a_1, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in \mathfrak{p}^{L}_{\mathfrak{A}}(v^{(k)}) \} & \text{otherwise.} \end{cases}$$

Notice that since L is the outcome of the translation of an Alloy specification, we can use that k above as the codified rank of the respective fields and variables.

Definition 3.2.7: Let $\mathfrak{a}$ be an Alloy instance. A proper PDOCFA $\mathfrak{F}$ is compatible with the instance $\mathfrak{a}$ if the relational instance $\mathfrak{p}_{\mathfrak{F}}$ (as defined in Def. 3.2.5), is correctly defined in $\mathfrak{F}$.

In Def. 3.2.7, *correctly defined* means that for each symbol s, $\mathfrak{p}_{\mathfrak{F}}(s)$ yields a relation in $\mathfrak{F}$. Lemmas assuring this correspondence are presented and proved in Appendix A.

This way of codifying higher-arity relations also impacts in the translation of the constants univ and iden. Both of them must be related to a binary relation that only holds urelements. We will denote such a relation by $\mathsf{iden}_{\mathsf{U}}$.

Another consequence is that some operations in PDOCFA must be modified in order to behave as expected for Alloy. Such operations are those whose result differs in arity from their parameters, namely: navigation and cartesian product. We define in PDOCFA two new operations denoted by $\bullet$ (for navigation) and $\boxtimes$ (for cartesian product) that preserve the previously given invariant. Before doing so, we introduce some notation.

Notation 3: In a proper PDOCFA the relations π and ρ defined by[3]

$$\sim(\mathsf{iden}\,\nabla\,\mathsf{univ}) \overset{\text{✍}}{=} \pi \quad \text{and} \quad \sim(\mathsf{univ}\,\nabla\,\mathsf{iden}) \overset{\text{✍}}{=} \rho$$

behave as projections with respect to the encoding of pairs induced by the injective function $\star$. Their semantics in a proper PDOCFA $\mathfrak{A}$ whose binary relations range over a set B, is

$$\mathcal{X}_{\mathfrak{A}}(\pi)\mathfrak{p}_{\mathfrak{A}} = \{\, \langle a \star b, a\rangle : a, b \in B \,\} \quad \text{and} \quad \mathcal{X}_{\mathfrak{A}}(\rho)\mathfrak{p}_{\mathfrak{A}} = \{\, \langle a \star b, b\rangle : a, b \in B \,\}$$

where $\mathfrak{p}_{\mathfrak{A}}$ is any instance for $\mathfrak{A}$.

The binary operation *cross* (denoted by $\otimes$) performs a parallel product. Its set-theoretical definition is given by

$$\mathcal{X}_{\mathfrak{A}}(e_1 \otimes e_2)\mathfrak{p}_{\mathfrak{A}} = \{\, \langle a \star c, b \star d\rangle : \langle a, b\rangle \in \mathcal{X}_{\mathfrak{A}}(e_1)\mathfrak{p}_{\mathfrak{A}} \text{ and } \langle c, d\rangle \in \mathcal{X}_{\mathfrak{A}}(e_2)\mathfrak{p}_{\mathfrak{A}} \,\}$$

where e_1 and e_2 are arbitrary PDOCFA expressions. In algebraic terms, operation cross is definable with the aid of fork via the equation

$$(\pi\,.\,e_1)\ \nabla\ (\rho\,.\,e_2) \overset{\text{✍}}{=} e_1 \otimes e_2$$

By $\mathsf{dom}(e)$ we denote the partial identity over the elements in e's domain. Similarly, by $\mathsf{ran}(e)$ we denote the partial identity over the elements in e's range. In algebraic terms, we have

$$(e\ .\ \sim e)\ \&\ \mathsf{iden} \overset{\text{✍}}{=} \mathsf{dom}(e), \quad \text{and} \quad (\sim e\ .\ e)\ \&\ \mathsf{iden} \overset{\text{✍}}{=} \mathsf{ran}(e)\ .$$

Definition 3.2.8:

$$e_1 \bullet e_2 \overset{\text{✍}}{=} \begin{cases} \mathsf{ran}(e_1.e_2) & \text{if } rank(e_1) = 1 \wedge rank(e_2) = 2 \\ \sim\pi\,.\,\mathsf{ran}(e_1.e_2)\,.\,\rho & \text{if } rank(e_1) = 1 \wedge rank(e_2) > 2 \\ \mathsf{dom}(e_1.e_2) & \text{if } rank(e_1) = 2 \wedge rank(e_2) = 1 \\ e_1.e_2 & \text{if } rank(e_1) = 2 \wedge rank(e_2) > 1 \\ e_1\,.\,\left(\mathsf{iden}\otimes^{rank(e_1)-3}\left((\mathsf{iden}\otimes e_2)\,.\,\pi\right)\right) & \text{if } rank(e_1) > 2 \wedge rank(e_2) = 1 \\ e_1\,.\,\left(\mathsf{iden}\otimes^{rank(e_1)-3}\left(\mathsf{iden}\otimes e_2\right)\right) & \text{if } rank(e_1) > 2 \wedge rank(e_2) > 1 \end{cases} \tag{3.10}$$

[3] We use the construction $a \overset{\text{✍}}{=} b$ to express that b will stand as an abbreviation of a or vice versa. The orientation should be clear by the context.

where $R \otimes^0 S = S$ and $R \otimes^{n+1} S = R \otimes (R \otimes^n S)$.

$$e_1 \boxtimes e_2 \triangleq \begin{cases} (e_1.univ).e_2 & \text{if } rank(e_1) = 1 \wedge rank(e_2) = 1 \\ (e_1.univ).(iden \nabla e_2) & \text{if } rank(e_1) = 1 \wedge rank(e_2) > 1 \\ \sim\pi \;.\; e_1 \otimes e_2 & \text{if } rank(e_1) = 2 \wedge rank(e_2) = 1 \\ \sim\pi \;.\; e_1 \otimes (iden \nabla e_2) & \text{if } rank(e_1) = 2 \wedge rank(e_2) > 1 \\ e_1 \;.\; \sim\pi \;.\; (iden \nabla e_2) & \text{if } rank(e_1) > 2 \wedge rank(e_2) = 1 \\ e_1 \;.\; \sim\pi \;.\; iden \otimes (iden \nabla e_2) & \text{if } rank(e_1) > 2 \wedge rank(e_2) > 1 \end{cases} \tag{3.11}$$

Following the previous example, if `ob` is an object atom (i.e., a unary Alloy relation of the form $\{ \, ob \, \}$ for some ob from signature `Object`), the navigation `contents.ob` produces as a result a binary relation contained in `Dir` $\times$ `Name`. Let us analyze what is the result of applying $\bullet$ on the PDOCFA representation of contents and ob. We obtain:

$$\begin{aligned}
&\mathcal{X}(\mathsf{contents} \bullet \mathsf{ob})\mathfrak{p} \\
= &\text{ (by Def. of } \bullet) \\
&\mathcal{X}(\mathsf{contents}.(\mathsf{iden} \otimes \mathsf{ob}).\pi)\mathfrak{p} \\
= &\text{ (by Def. of ``.'')} \\
&\{\langle d, a\rangle : \text{some } x : \mathtt{Name}, y : \mathtt{Object} \mid \langle d, x \star y\rangle \text{ in } \mathsf{contents} \wedge \\
&\quad \text{some } x' : \mathtt{Name}, y' : \mathtt{Object} \mid \langle x \star y, x' \star y'\rangle \text{ in } \mathsf{iden} \otimes \mathsf{ob} \wedge \\
&\quad \langle x' \star y', a\rangle \in \pi\} \\
= &\text{ (because } \langle x, x'\rangle \in \mathsf{iden} \wedge \langle y, y'\rangle \in \mathsf{ob} \subseteq \mathsf{iden}) \\
&\{\langle d, a\rangle : \text{some } x : \mathtt{Name}, y : \mathtt{Object} \mid \langle d, x \star y\rangle \text{ in } \mathsf{contents} \wedge \\
&\quad \langle y, y\rangle \in \mathsf{ob} \wedge \langle x \star y, a\rangle \in \pi\} \\
= &\text{ (by Def. of } \pi, \text{ is } x = a) \\
&\{\langle d, a\rangle : \text{some } y : \mathtt{Object} \mid \langle d, a \star y\rangle \text{ in } \mathsf{contents} \wedge \langle y, y\rangle \in \mathsf{ob}\} \\
= &\text{ (due to the relationship between } \mathsf{contents} \text{ and } \mathtt{contents}, \text{ and } \mathsf{ob} \text{ and } \mathtt{ob}) \\
&\{\langle d, a\rangle : \text{some } y : \mathtt{Object} \mid \langle d, a, y\rangle \text{ in } \mathtt{contents} \;\&\&\; y \in \mathtt{ob}\} \\
= &\text{ (because } \mathtt{ob} \text{ is an atom from } \mathtt{Object}) \\
&\{\langle d, a\rangle : \langle d, a, \mathtt{ob}\rangle \text{ in } \mathtt{contents}\}
\end{aligned}$$

It is worth emphasizing that, while Alloy formulas and PDOCFA formulas are close, there are still differences between the formalisms. The differences from the Alloy language arise when we need to prove properties of $\bullet$ or $\boxtimes$ that require using the underlying fork algebra definition that includes the operator ∇. We expect the number of properties involving the definition of such operators to be small compared to the complete proof. For example, in the case study we are reporting, 25 out of 60 proved lemmas required dealing with the low-level representation of $\bullet$. Yet 18 out of these 25 properties relate to properties of $\bullet$ that may be reused along other proofs. For example, among these 18 properties the following general properties are included:

```
all A:set univ, B:set univ, R:set(A->B), S:set(A->B), a:A |
   R in S implies a.R in a.S
all A:set univ, B:set univ, R:set(A->B), S:set(A->B) |
   (R.B)+(S.B) = (R+S).B
all A:set univ, B:set univ, C:set univ, W:set(A->(B->C)), a:A |
   a.W in B->C
```

We include a library with those properties of $\bullet$ and $\boxtimes$ that we consider general and useful.

Constraining PDOCFA Quantifiers to Points

When we write an Alloy formula such as

$$\mathsf{all\ d, d' : Domain \mid d.space! = d'.space => d! = d'}$$

quantified variables d and d′ range over `Domain` objects. There is a single signature in PDOCFA, namely, the one that holds all the relations. Therefore, given an algebra in PDOCFA, quantifiers range over all the relations in the domain of the algebra. Hence, while the Alloy operations have an almost direct counterpart in PDOCFA, quantified formulas do not.

This is when the notion of point become necessary. A *point* is a relation of the form $\{\langle a, a\rangle\}$. We constrain the PDOCFAs used to interpret the outcomes of the translation of Alloy specifications, to be "point-dense" [Maddux (1991)]. Point-density requires set $\mathbf{R}_{\mathfrak{A}}$ to have plenty of these relations. More formally speaking, for each nonempty relation I contained in the identity relation, there must be a point $p \in R$ satisfying $p \subseteq I$.

We then associate an Alloy singleton $\{a\}$ with the point $\{\langle a, a\rangle\}$. As stated in Def. 3.2.5, we will associate Alloy signatures with partial identities in PDOCFA.

Notation 4: We can characterize points as nonempty binary relations that satisfy the property $x.\mathsf{univ}.x \subseteq \mathsf{iden}$. If we denote the inclusion relation by "in" (as in Alloy), the predicate "point" defined by

$$\mathsf{point}(p) \overset{✍}{=} p \mathrel{!=} \mathsf{none}\ \&\&\ p.\mathsf{univ}.p \text{ in } \mathsf{iden}$$

characterizes those relations that are points.

We can then map an Alloy formula of the form

$$\mathsf{all\ a : A} \mid \alpha$$

to a PDOCFA formula of the form

$$\mathsf{all\ a \mid (point(a)\ \&\&\ a\ in\ iden_A) => } \alpha' \tag{3.12}$$

where α' is the PDOCFA formula resulting from the translation of the Alloy formula α. In order to retain the similarity between Alloy formulas and their counterparts, we will introduce the following notation:

$$\text{all a} \mid (\text{point(a) \&\& a in iden}_{\text{A}}) \Rightarrow \alpha' \quad \overset{✍}{=} \quad \text{all a : A} \mid \alpha'$$

Notice that the above abbreviation equates (up-to translation of terms) the source Alloy formulas and their translation to PDOCFA.

As showed at the beginning of this chapter, there are several ways of stating a restriction on a quantified variable in Alloy, but all of them can be adequately expressed in PDOCFA. Further, Alloy syntax allows higher-order quantifications, although the Alloy Analyzer is rarely able to manipulate them. Following the idea presented, all those kinds of quantifications can be expressed in PDOCFA, as we will show in section 3.2.3.

Alloy Integers in PDOCFA

Recall from Ch. 2 that Alloy integers are defined relative to a user-provided bound [Jackson (2012)] called the *bit width*. We will denote by $+_{\mathbf{bw}}$ the arithmetic sum relative to a bit width *bw*. Numeric atoms may appear in relations like regular atoms do. In fact, they both have the same status.

To support Alloy integers in Dynamite, we enrich PDOCFA theories with new constants, functions, predicates, and their corresponding axioms. We add a new partial identity $iden_{\mathbb{Z}}$, which models the set of Alloy integer atoms in the range $\left[-2^{\mathbf{bw}-1}, 2^{\mathbf{bw}-1}-1\right]$ determined by the bit width. In our theory, $iden_{\mathbb{Z}}$ is a set of urelements, and 0 and **bw** are constants that denote the integer value 0 and the bit width, respectively. Notice that while 0 is always contained in $iden_{\mathbb{Z}}$, **bw** may not be when $\mathbf{bw} \in \{1, 2\}$. Therefore, we will focus on the general case ($\mathbf{bw} > 2$), and come back to the cases in which $\mathbf{bw} \in \{1, 2\}$ after the presentation of the general case. Axiomatically,

$$\begin{gathered}\text{iden}_{\mathbb{Z}} \text{ in iden}_{\text{U}} \\ \text{point}(0) \\ \text{point(bw)} \\ 0 \text{ in iden}_{\mathbb{Z}} \\ \text{bw in iden}_{\mathbb{Z}}\end{gathered} \tag{3.13}$$

We introduce a binary predicate symbol $<$ which stands for a linear order with endpoints, over $iden_{\mathbb{Z}}$.

In order to simplify the notation, quantifications over $\mathbb{Z}$ are indeed quantifications over points contained in $iden_{\mathbb{Z}}$. For example:

$$\text{all a} \mid (\text{Point(a) \&\& a in iden}_{\mathbb{Z}}) \Rightarrow \alpha' \quad \overset{✍}{=} \quad \text{all a} : \mathbb{Z} \mid \alpha'$$

$$\text{some a} \mid (\text{Point(a) \&\& a in iden}_{\mathbb{Z}}) \text{ \&\& } \alpha' \quad \overset{✍}{=} \quad \text{some a} : \mathbb{Z} \mid \alpha'$$

We will denote by $\mathsf{Max}(x)$ the integer unary predicate $!\mathsf{some}\ a : \mathbb{Z} \mid x < a$. A predicate $\mathsf{Min}(x)$ is symmetrically defined. Binary predicate $<$ is characterized by the axioms

$$\mathsf{all\ a,b} : \mathbb{Z} \mid \mathsf{a} < \mathsf{b} \mid\mid \mathsf{b} < \mathsf{a} \mid\mid \mathsf{a} = \mathsf{b}$$

$$\mathsf{all\ a} : \mathbb{Z} \mid !(\mathsf{a} < \mathsf{a})$$
$$\mathsf{all\ a,b,c} : \mathbb{Z} \mid (\mathsf{a} < \mathsf{b}\ \&\&\ \mathsf{b} < \mathsf{c}) \Rightarrow \mathsf{a} < \mathsf{c}$$
$$\mathsf{some\ a} : \mathbb{Z} \mid \mathsf{Min(a)}$$
$$\mathsf{some\ a} : \mathbb{Z} \mid \mathsf{Max(a)}\ .$$

We next introduce the unary function succ, which models the successor function (+1) according to 2's complement arithmetic:

$$\mathsf{all}\ a, b : \mathbb{Z} \mid (\mathsf{Max}(b)\ \&\&\ a < b) \Rightarrow a < \mathsf{succ}(a),$$

$$\mathsf{all}\ a : \mathbb{Z} \mid !\mathsf{some}\ b : \mathbb{Z} \mid a < b\ \&\&\ b < \mathsf{succ}(a),$$
$$\mathsf{all}\ a, b : \mathbb{Z} \mid (\mathsf{Max}(a)\ \&\&\ \mathsf{Min}(b)) \Rightarrow \mathsf{succ}(a) = b.$$

Having defined the successor of a numeric point n, defining its predecessor (noted $\mathsf{prev}(n)$), additive inverse ($-_{\mathsf{bw}}n$), addition of numeric points n and m ($n +_{\mathsf{bw}} m$), subtraction ($n -_{\mathsf{bw}} m$), multiplication ($n \cdot_{\mathsf{bw}} m$), integer division ($n /_{\mathsf{bw}} m$), the remainder of the integer division ($n\ \%_{\mathsf{bw}}\ m$) and power ($[n^m]_{\mathsf{bw}}$) becomes an easy exercise. We present the definition of addition, as an example:

$$\mathsf{all}\ a : \mathbb{Z} \mid a +_{\mathsf{bw}} 0 = a, \quad \mathsf{all}\ a, b : \mathbb{Z} \mid a +_{\mathsf{bw}} \mathsf{succ}(b) = \mathsf{succ}(a +_{\mathsf{bw}} b).$$

From now on, when referring to these operations we will omit the subscript bw where possible to improve readability.

Once the operations have been defined, we can axiomatize the proper value of the endpoints, as well as remark that bw must be greater than 2:

$$\mathsf{Min}(-2^{\mathsf{bw}-1})$$
$$\mathsf{Max}(2^{\mathsf{bw}-1} - 1)$$
$$\mathsf{bw} > \mathsf{succ}(\mathsf{succ}(0))$$

Supporting the Alloy cardinality operator # requires the addition of a new function card to PDOCFA. As expected, points have cardinality 1. The cardinality of an arbitrary relation is defined by formulas that, for finite relations, make card return the number of tuples[4]:

$$\mathsf{all}\ r \mid \mathsf{Point}(r)\ \Rightarrow\ \mathsf{card}(r) = \mathsf{succ}(0),$$

[4] Unary predicate Some characterizes nonempty relations. $\epsilon(A)$ retrieves a pair contained in A, and $\setminus$ stands for set difference.

$$\mathsf{card}(\emptyset) = 0, \quad \mathsf{all}\ r \mid \ \mathsf{Some}(r) \Rightarrow \mathsf{card}(r) = \mathsf{succ}(0)\ +_{\mathbf{bw}} \mathsf{card}(r \backslash \epsilon(r))\ .$$

Since numeric constants in a specification cannot take values off the range determined by bw, for each numeric constant symbol c in the Alloy specification we add axioms:

$$-2^{\mathsf{bw}-1} \leq c \quad \text{and} \quad c \leq 2^{\mathsf{bw}-1} - 1\ .$$

Given an Alloy integer expression e, the Alloy expression `Int[`e`]` denotes the integer atom holding the integer value of e. Conversely, the Alloy function `int` returns the sum of the integer values corresponding to the integer atoms included in a given Alloy expression. For example, `Int[2]` denotes the numeric atom corresponding to the integer 2, which in turn is the result of `int[Int[2]]`.

In PDOCFA we make no distinction between integer values and integer atoms. We will use points contained in $iden_{\mathbb{Z}}$ to represent both kinds on entities. We model function `Int` in PDOCFA with an unary function Int, which is defined as the identity. We also introduce the function int, also unary, which models `int`. Axioms

$$\mathsf{all}\ a : \mathbb{Z} \mid \mathsf{int}\,(a) = a$$

$$\mathsf{all}\ a : \mathsf{iden}_{\mathsf{U}} - \mathsf{iden}_{\mathbb{Z}} \mid \mathsf{int}\,(a) = 0$$
$$\mathsf{int}\,(\emptyset) = 0$$

state that int behaves as the identity on integer atoms, and returns 0 for non integer atoms or the empty relation. For more complex relational expressions, int must add the values of the integer points contained in the expression. This is captured by the following axiom:

$$\mathsf{all}\ r \mid \mathsf{Some}(r) \Rightarrow \mathsf{int}\,(r) = \mathsf{int}\,(\epsilon(r)) +_{\mathsf{bw}} \mathsf{int}\,(r \backslash \epsilon(r))\,.$$

To support the functionality provided by the Alloy construction sum, that allows the user to express a summation of Alloy numeric expressions with free variables ranging over sets, we will include specific functions and axioms for every such construction present in the user specification. If the expression

$$\mathsf{sum}\ v_1{:}E_1, \cdots, v_n{:}E_n \mid \nu$$

(where ν is a numeric expression with n free variables: v_1 to v_n) appears in the Alloy specification, we will include in the final PDOCFA specification: n function symbols sum^1_ν, sum^2_ν, $\cdots$, sum^n_ν and the n axioms

$$\begin{aligned}
&\mathsf{all}\ \mathsf{X}_1, \cdots, \mathsf{X}_n \mid \big(\mathsf{X}_1 = \emptyset \Rightarrow \mathsf{sum}^1_\nu(\mathsf{X}_1, \cdots, \mathsf{X}_n) = 0\big)\ \&\&\ \big(\mathsf{Some}(\mathsf{X}_1) \Rightarrow \\
&\quad \mathsf{sum}^1_\nu(\mathsf{X}_1, \cdots, \mathsf{X}_n) = \mathsf{sum}^1_\nu(\mathsf{X}_1 \backslash \epsilon(\mathsf{X}_1), \cdots, \mathsf{X}_n) +_{\mathsf{bw}} \mathsf{sum}^2_\nu(\epsilon(\mathsf{S}_1), \mathsf{X}_2, \cdots, \mathsf{S}_n)\big) \\
&\qquad\qquad\vdots \\
&\mathsf{all}\ \mathsf{x}_1, \cdots, \mathsf{X}_n \mid \big(\mathsf{X}_n = \emptyset \Rightarrow \mathsf{sum}^n_\nu(\mathsf{x}_1, \cdots, \mathsf{X}_n) = 0\big)\ \&\&\ \big(\mathsf{Some}(\mathsf{X}_n) \Rightarrow \\
&\quad \mathsf{sum}^n_\nu(\mathsf{x}_1, \cdots, \mathsf{X}_n) = \mathsf{sum}^n_\nu(\mathsf{x}_1, \cdots, \mathsf{X}_n \backslash \epsilon(\mathsf{X}_n)) +_{\mathsf{bw}} \mathbf{T}^{[\![v_1 \mapsto \mathsf{x}_1, \cdots, v_n \mapsto \epsilon(\mathsf{X}_n)]\!]}(\nu)\big)
\end{aligned}$$

where $\mathbf{T}^{[\![v_1 \mapsto \mathsf{x}_1, \cdots, v_n \mapsto \epsilon(\mathsf{X}_n)]\!]}(\nu)$ is the result of applying the usual term translation $\mathbf{T}$ (to be detailed below) on ν, but replacing each Alloy variable v_i by the corresponding PDOCFA expression e_i, as stated by every $v_i \mapsto e_i$.

The theories that model Alloy integers in PDOCFA when $\mathsf{bw} \in \{1, 2\}$, are obtained by adequately instantiating the above theory, while at the same time removing axioms (3.13) and (3.14). For instance, for $\mathsf{bw} = 1$, the axiomatization of the end points becomes $\mathsf{Min}(\mathsf{succ}(0))$ and $\mathsf{Max}(0)$. Notice that, in this case, $\mathsf{succ}(0)$ is indeed -1.

For PDOCFA models in which the set of integer points is finite, the theory correctly captures Alloy's semantics. Notice that since the theory admits arbitrarily large finite models, by compactness it must admit infinite models as well. We leave the study of such models as further work.

Unlike [Ulbrich et al. (2012)], which departs from Alloy's semantics and considers the standard infinite model for integers, we consider 2's complement arithmetic on integers representable using a finite bit width. For Dynamite this is not an optional feature of the language, but rather the only possible choice. In Section 4.2 we will present the main features of Dynamite. One such feature is the use of the Alloy Analyzer to look for counterexamples of properties being verified. To be useful, counterexamples provided by the Alloy Analyzer must agree with Alloy's semantics as captured in Dynamite's calculus. Otherwise, counterexamples generated by the Alloy Analyzer would fail as counterexamples for the property being verified with the aid of Dynamite.

The Point-dense Omega Closure Fork Algebras

We now introduce a larger class of algebras as the class of models of a finitely axiomatized theory. These algebras, called point-dense omega closure fork algebras (we will denote the class by PDOCFA) are closely related (as will be shown in Thm. 3.2.10) to their "proper" counterpart. In order to present the theory we will present the axioms and the proof rules.

The axioms and inference rules for the calculus are given in the following definition.

Definition 3.2.9: (PDOCFA calculus) The calculus for point-dense omega closure fork algebras is characterized by the following axioms and proof rules:

1. Axioms for Boolean algebras characterizing $+$, $\&$, $^{-}$, none and univ:

$$
\begin{aligned}
\text{all } x &\mid x + x = x,\\
\text{all } x &\mid x \,\&\, x = x,\\
\text{all } x, y &\mid x + y = y + x,\\
\text{all } x, y &\mid x \,\&\, y = y \,\&\, x,\\
\text{all } x, y, z &\mid x + (y + z) = (x + y) + z,\\
\text{all } x, y, z &\mid x \,\&(y \,\&\, z) = (x \,\&\, y)\& \, z,\\
\text{all } x, y, z &\mid x + (x \,\&\, y) = x,\\
\text{all } x, y, z &\mid x \,\&(x + y) = x,\\
\text{all } x, y, z &\mid x \,\&(y + z) = (x \,\&\, y) + (x \,\&\, y),\\
\text{all } x, y, z &\mid x +(y \,\&\, z) = (x + y)\&(x + y),\\
\text{all } x &\mid x \,\&\, \text{none} = \text{none},\\
\text{all } x &\mid x + \text{univ} = \text{univ},\\
\text{all } x &\mid x \,\&\, \overline{x} = \text{none},\\
\text{all } x &\mid x + \overline{x} = \text{univ}.
\end{aligned}
$$

2. Formulas defining composition of binary relations, transposition, reflexive–transitive closure and the identity relation:

$$
\begin{aligned}
\text{all } x, y &\mid x.\,(y.z) = (x.y)\,.z,\\
\text{all } x &\mid x.\text{iden} = \text{iden}.x = x,\\
\text{all } x, y, z &\mid (x.y)\,\&z = \text{none} \Leftrightarrow (z.\,\sim y)\,\&x = \text{none} \Leftrightarrow (\sim x.z)\,\&y = \text{none},\\
\text{all } x &\mid *x = \text{iden} + (x.*x),\\
\text{all } x, y &\mid *x.y.\text{univ in } (y.\text{univ}) + \big(*x.(\overline{y.\text{univ}} \,\&\, (x.y.\text{univ}))\big).
\end{aligned}
$$

3. Formulas defining the operator ∇:

$$
\begin{aligned}
&\text{all } x, y \mid x\nabla y = (x.\sim \pi) \,\&\, (y.\sim \rho),\\
&\text{all } x, y \mid (x\nabla y)\,.\sim (w\nabla z) = (x.\sim w) \,\&\, (y.\sim z),\\
&\pi\nabla\rho \text{ in iden}.
\end{aligned}
$$

4. A formula enforcing point-density:

$$\text{all } x \mid (x \mathrel{!=} \text{none} \,\&\&\, x \text{ in iden}) \Rightarrow (\text{some } p \mid \text{Point}(p) \,\&\&\, p \text{ in } x)\,.$$

5. Term $\overline{\text{univ}.\,(\text{univ}\nabla\text{univ})} \,\&\, \text{iden}$ (to be abbreviated as iden_U) defines a partial identity on the set of urelements. Then, the following formula forces the existence of a nonempty set of urelements:

$$\text{univ}.\text{iden}_U.\text{univ} = \text{univ}.$$

6. The axioms supporting the numeric part of the translation, showed in the previous section.

The inference rules for the closure fork calculus are those for classical first-order logic (choose your favorite ones), plus the following equational (but infinitary) proof rule for reflexive-transitive closure (given $i > 0$, by x^i we denote the relation inductively defined as follows: $x^1 = x$, and $x^{i+1} = x.x^i$):

$$\frac{\vdash \text{iden in } y \quad x^i \text{ in } y \vdash x^{i+1} \text{ in } y}{\vdash *x \text{ in } y} \;(\omega\text{−}Rule)$$

The axioms and rules given above define a class of models. Proper PDOCFA belong to this class, but there might be models for the axioms that are not proper PDOCFA. Fortunately, the following theorem, which follows from [Frias et al. (1997), (Frias, 2002, Thm. 4.2) and (Maddux, 1991, Thm. 52)], states that if a model is not a proper PDOCFA, then it is isomorphic to one.

Theorem 3.2.10: Every PDOCFA $\mathfrak{A}$ is isomorphic to a proper PDOCFA $\mathfrak{B}$. Moreover, there exist relations $\{\langle a_0, a_0\rangle\}, \ldots, \{\langle a_i, a_i\rangle\}\ldots$ (possibly infinitely many of them) that belong to $\mathfrak{B}$, such that

$$iden = \{\langle a_0, a_0\rangle, \ldots, \langle a_i, a_i\rangle, \ldots\}$$

While this theorem is important in itself, since it implies that the calculus is complete with respect to the properties valid in proper PDOCFAs, it is necessary in order to prove theorems on the appropriateness of the deductive mechanism we will provide for Alloy in Section 3.2.3.

3.2.3 Interpretability of Alloy in PDOCFA

One of the (main) goals of this thesis is to present a complete deductive mechanism for Alloy. In order to fulfill this task we will prove an interpretation theorem of Alloy specifications as PDOCFA theories. An interpretation theorem of Alloy in PDOCFA (as described in the introduction to Section 3.2), allows us to map semantic entailment in Alloy to deductions in PDOCFA. Said result allows us to use the calculus for PDOCFA in the following way. If we want to prove a given assertion α in an Alloy model (specification) M, we construct a $\mathscr{L}$-PDOCFA L compatible with M, using L we translate α and the facts in M to PDOCFA formulas α' and axioms M', and prove that α' follows from M' according to the PDOCFA calculus.

The initial part of the translation process was already discussed: based on the user Alloy specification S we construct a compatible language L by relating fresh constant, function and predicate symbols to every signature, field, function and predicate defined in S. Then, we collect all the facts in S and translate them into PDOCFA axioms. Furthermore, all the implicit restrictions imposed, for example, by the declarations of signatures and fields, are also stated in PDOCFA terms and incorporated to the resulting theory as new axioms. Additionally, the PDOCFA theory must possess all the axioms mentioned when we presented the calculus and in the Alloy integers representability section. The assertions declared in the original specification S are included also as conjectures, and available to the user to be used as lemmas (to be proven later).

As we discussed in Chapter 1, the idea of mapping Alloy to an expressive-enough formalism in which to carry proofs on is not entirely new. It has already been done for instance with Prioni [Arkoudas et al. (2004)]. The (essential) advantage of the mapping we propose in this thesis is that the resulting formalism is extremely close to Alloy, and therefore easier to grasp by standard Alloy users. While this feature may be useless in the context of fully automated tools (for which the language target of the translation may be ignored by the user), it is of utter importance for user-guided tools. In fact, our $\mathscr{L}$-PDOCFAs are so close to Alloy that they allow us to only present Alloy formulas and expressions to the user during the whole proving process. In the remaining parts of this section we give a proof of the interpretability theorem.

The main part of an interpretability theorem is a mapping from Alloy formulas to formulas in the language of PDOCFA. The mapping is defined in two stages, since Alloy terms must be mapped as well. We will present maps **T** (mapping terms), and **F** (mapping formulas). As reader will see, both mappings participate in each other's definition.

Comprehension expressions

The *comprehension* expressions allow the user to define anonymous relations with specific restrictions imposed on its tuples. We try to keep the target language as simple as possible, so in order to cope with the translation of such expressions, we add to the target language L: *(1)* a fresh constant symbol for each such definition in the source Alloy specification, and *(2)* a new axiom that states the restriction imposed in the definition of the comprehension expression. Let us see an example. Suppose we have an Alloy comprehension expression such as:

```
{ d: Dir | d.contents != none }
```

with relation contents from signature `Dir`. In order to translate it, we add a fresh constant to the target PDOCFA language, let's call it C_1, and the following axiom to the resulting theory:

$$\mathsf{all\ d}\ |\ \mathsf{d\ in\ C_1} \Leftrightarrow \mathsf{d} \bullet \mathsf{contents}\ ! =\ \mathsf{none}$$

Mapping of Alloy terms to PDOCFA expressions

The following definition introduces the mapping for terms.

Definition 3.2.11: (Translation of Alloy terms) Let M be an Alloy specification and L a PDOCFA language compatible with it. The function

$$\mathbf{T}_{M \cdot L} : \mathscr{E}_M^{\mathsf{Alloy}} \to \mathscr{E}_L^{\mathsf{PDOCFA}}$$

maps Alloy terms to expressions in the language L. The detailed definition is given in table 3.4. As usual, we will omit the subscripts wherever possible.

e	$\mathbf{T}(e)$
none	none
univ	$\mathsf{iden}_{\mathsf{U}}$
iden	$\mathsf{iden}_{\mathsf{U}}$
Being s a signature:	
s	$\mathsf{iden}_{\mathsf{s}}$
Being f a field:	
f	f

e	$\mathbf{T}(e)$
Int	$\mathsf{iden}_{\mathbb{Z}}$
$\mathsf{Int}[n]$	$\mathsf{Int}\,(\mathbf{T}(n))$
$\sim e$	$\sim\mathbf{T}(e)$
$*e$	$*\mathbf{T}(e)$
$\hat{}e$	$\mathbf{T}(e).*\mathbf{T}(e)$
$\#e$	$\mathsf{card}(\mathbf{T}(e))$
v_i	V_i

e	$\mathbf{T}(e)$
$e_1 + e_2$	$\mathbf{T}(e_1) + \mathbf{T}(e_2)$
$e_1 \ \& \ e_2$	$\mathbf{T}(e_1) \ \& \ \mathbf{T}(e_2)$
$e_1 - e_2$	$\mathbf{T}(e_1) \ \& \ \overline{\mathbf{T}(e_2)}$
$e_1 \ . \ e_2$	$\mathbf{T}(e_1) \ \bullet \ \mathbf{T}(e_2)$
$e_1 <: e_2$	$\mathbf{T}(e_1) \ \& \ (\mathbf{T}(e_2).\mathsf{univ}) \quad \overset{\text{✍}}{=} \quad \mathbf{T}(e_1) <: \mathbf{T}(e_2)$
$e_1 :> e_2$	$\mathbf{T}(e_1) \ \& \ (\mathsf{univ}.\mathbf{T}(e_2))$
$e_1 ++ e_2$	$\big((\mathsf{dom}(\mathbf{T}(e_1)) - \mathsf{dom}(\mathbf{T}(e_2)))\ <: \mathbf{T}(e_1)\big) + \mathbf{T}(e_2)$
$e_1 -> e_2$	$\mathbf{T}(e_1) \ \boxtimes \ \mathbf{T}(e_2)$
$e_1[e_2]$	$\begin{cases} \mathbf{T}(e_1)(\mathbf{T}(e_2)) & \text{if } e_1 \text{ is a function symbol} \\ \mathbf{T}(e_2) \ \bullet \ \mathbf{T}(e_1) & \text{otherwise} \end{cases}$
$\alpha \ [=> \mid \mathsf{implies} \ ] \ e_1 \ \mathsf{else} \ e_2$	$\mathbf{F}(\alpha) => \mathbf{T}(e_1) \ \mathsf{else} \ \mathbf{T}(e_2)$
$\mathsf{let} \ v_1{=}e_1, \cdots, v_n{=}e_n \mid e$	$\mathsf{let} \ \mathbf{T}(v_1){=}\mathbf{T}(e_1), \cdots, \mathbf{T}(v_n){=}\mathbf{T}(e_n) \mid \mathbf{T}(e)$
$\{v_1{:}e_1, \cdots, v_n{:}e_n \mid \alpha\}$	C_i where C_i is a fresh constant symbol, and i is the count of comprehension expressions formerly seen.
Being n a positive numeric constant:	
n	$\begin{cases} 0 & \text{if } n = 0 \\ \mathsf{succ}(0) & \text{if } n = 1 \\ \mathsf{succ}(\mathsf{succ}(0)) & \text{if } n = 2 \\ \vdots & \end{cases}$
$\# \ e$	$\mathsf{card}(\mathbf{T}(e))$
$-ne$	$-_{\mathsf{bw}}\mathbf{T}(ne)$
$[e.\mathsf{sum} \mid \mathsf{int}[e]]$	$\mathsf{int}\,(\mathbf{T}(e))$
$ne_1 \ \mathsf{add} \ ne_2$	$\mathbf{T}(ne_1) +_{\mathsf{bw}} \mathbf{T}(ne_2)$
$ne_1 \ \mathsf{sub} \ ne_2$	$\mathbf{T}(ne_1) -_{\mathsf{bw}} \mathbf{T}(ne_2)$
$ne_1 \ \mathsf{mul} \ ne_2$	$\mathbf{T}(ne_1) \ \cdot_{\mathsf{bw}} \ \mathbf{T}(ne_2)$
$ne_1 \ \mathsf{div} \ ne_2$	$\mathbf{T}(ne_1) \ /_{\mathsf{bw}} \ \mathbf{T}(ne_2)$
$ne_1 \ \mathsf{rem} \ ne_2$	$\mathbf{T}(ne_1) \ \%_{\mathsf{bw}} \ \mathbf{T}(ne_2)$
$\mathsf{sum} \ v_1{:}d_1, \cdots, v_n{:}d_n \mid ne$	$\mathsf{sum}^1_{ne}(\mathbf{T}(d_1), \cdots, \mathbf{T}(d_n))$

Tab. 3.4: Definition of mapping of Alloy terms to PDOCFA expressions.

Translation of Alloy formulas

Once the translation of terms has been presented, we introduce the translation from Alloy formulas to PDOCFA formulas. The translation differs from the one presented in [Frias et al. (2004)] in that the target of the translation is a first-order language rather than an equational language, and therefore it is no longer necessary to encode quantified variables because they are kept explicit. This will greatly improve the readability of the translated formulas by Alloy users.

Prior to presenting the formal definition of the formula mapping, we introduce more notation with the aim of improving its readability.

Notation 5: We will use predicates for testing cardinality analogous to those provided by Alloy.

$$\begin{aligned}
\mathsf{no}(e) &\overset{✍}{=} e\!= \mathsf{none}\\
\mathsf{some}(e) &\overset{✍}{=} e! = \mathsf{none}\\
\mathsf{one}(e) &\overset{✍}{=} \mathsf{card}(e)\!= \mathsf{succ}(0)\\
\mathsf{lone}(e) &\overset{✍}{=} \mathsf{no}(e) \mid\mid \mathsf{one}(e)
\end{aligned}$$

The operator of inclusion, which was already used, is in fact an abbreviation.

$$e_1 \ \mathsf{in}\ e_2 \ \overset{✍}{=}\ e_2 = e_1 + e_2$$

Also, we will use abbreviations for the negation of equality, inclusion and numeric comparison.

$$\begin{aligned}
e_1 \ != e_2 &\overset{✍}{=} !(e_1 = e_2)\\
e_1 \ !\mathsf{in}\ e_2 &\overset{✍}{=} !(e_1 \ \mathsf{in}\ e_2)\\
e_1 \ !< e_2 &\overset{✍}{=} !(e_1 < e_2)\\
e_1 \ !> e_2 &\overset{✍}{=} !(e_1 > e_2)
\end{aligned}
\qquad
\begin{aligned}
e_1 \leq e_2 &\overset{✍}{=} e_1 {<} e_2 \mid\mid e_1{=}e_2\\
e_1 > e_2 &\overset{✍}{=} !(e_1 \leq e_2)\\
e_1 \geq e_2 &\overset{✍}{=} e_1 {>} e_2 \mid\mid e_1{=}e_2\\
e_1 \ !\leq e_2 &\overset{✍}{=} !(e_1 \leq e_2)\\
e_1 \ !\geq e_2 &\overset{✍}{=} !(e_1 \geq e_2)
\end{aligned}$$

Definition 3.2.12: (Translation of Alloy formulas) Let M be an Alloy specification and L a PDOCFA language compatible with it. The function

$$\mathbf{F}_{M \cdot L} : \Gamma_M^{\mathsf{Alloy}} \to \Gamma_L^{\mathsf{PDOCFA}}$$

maps Alloy formulas to PDOCFA formulas in L. The detailed definition of $\mathbf{F}_{M \cdot L}$ can be seen in table 3.5. For the sake of clarity, we will be omitting the subscripts when possible.

Notice that the result of the translations of Alloy quantifications over atoms, although different from those explained in formula 3.12, are equivalent to them.

We next will prove the following completeness theorem (recall that the turnstile symbol $\vdash$ notes the derivability relation in the calculus of PDOCFAs).

α	$\mathbf{F}(\alpha)$
no e	no($\mathbf{T}(e)$)
some e	some($\mathbf{T}(e)$)
lone e	lone($\mathbf{T}(e)$)
one e	one($\mathbf{T}(e)$)
$e_1 = e_2$	$\mathbf{T}(e_1) = \mathbf{T}(e_2)$
e_1 [!$\mid$not] $= e_2$	$\mathbf{T}(e_1)$!= $\mathbf{T}(e_2)$
e_1 in e_2	$\mathbf{T}(e_1)$ in $\mathbf{T}(e_2)$
e_1 [!$\mid$not] in e_2	$\mathbf{T}(e_1)$!in $\mathbf{T}(e_2)$
$n_1 = n_2$	$\mathbf{T}(n_1) = \mathbf{T}(n_2)$
n_1 [!$\mid$not] $= n_2$	$\mathbf{T}(n_1)$!= $\mathbf{T}(n_2)$
$n_1 < n_2$	$\mathbf{T}(n_1) < \mathbf{T}(n_2)$
n_1 [!$\mid$not] $< n_2$	$\mathbf{T}(n_1)$!< $\mathbf{T}(n_2)$
$n_1 <= n_2$	$\mathbf{T}(n_1) \leq \mathbf{T}(n_2)$
n_1 [!$\mid$not] $= n_2$	$\mathbf{T}(n_1)$!$\leq$ $\mathbf{T}(n_2)$
$n_1 > n_2$	$\mathbf{T}(n_1) > \mathbf{T}(n_2)$
n_1 [!$\mid$not] $> n_2$	$\mathbf{T}(n_1)$!> $\mathbf{T}(n_2)$
$n_1 >= n_2$	$\mathbf{T}(n_1) \geq \mathbf{T}(n_2)$
n_1 [!$\mid$not] $>= n_2$	$\mathbf{T}(n_1)$!$\geq$ $\mathbf{T}(n_2)$
!α	!α
$\alpha \Vert \beta$	$\alpha \Vert \beta$
α && β	α && β
α <=> β	$\alpha \Leftrightarrow \beta$
α => β	$\alpha \Rightarrow \beta$
let $v = e \mid \alpha$	let $\mathbf{T}(v) = \mathbf{T}(e) \mid \mathbf{F}(\alpha)$
let $v = \beta \mid \alpha$	let $\mathbf{T}(v) = \mathbf{F}(\beta) \mid \mathbf{F}(\alpha)$
let $v = n \mid \alpha$	let $\mathbf{T}(v) = \mathbf{T}(n) \mid \mathbf{F}(\alpha)$
let $n_1{=}x_1, \cdots, n_m{=}x_m \mid \alpha$	$\mathbf{F}$(let $n_1{=}x_1 \mid$ let $n_2{=}x_2, \cdots, n_m{=}x_m \mid \alpha$)
α => β else δ	$\mathbf{F}(\alpha)$ => $\mathbf{F}(\beta)$ else $\mathbf{F}(\delta)$
$\{\alpha_1 \cdots \alpha_n\}$	$\mathbf{F}(\alpha_1)$ && $\cdots$ && $\mathbf{F}(\alpha_n)$
all $v : d \mid \alpha$	all $v \mid \mathbf{F}((v$ ind' && $\mathcal{R}(v,d))$ => $\alpha)$ $\overset{✍}{=}$ all $v{:}\mathbf{T}(d) \mid \mathbf{F}(\alpha)$
some $v : d \mid \alpha$	some $v \mid \mathbf{F}(v$ ind' && $\mathcal{R}(v,d)$ && $\alpha)$ $\overset{✍}{=}$ some $v{:}\mathbf{T}(d) \mid \mathbf{F}(\alpha)$
no $v : d \mid \alpha$	!$\mathbf{F}$(some $v : d \mid \alpha$) $\overset{✍}{=}$ no $v{:}\mathbf{T}(d) \mid \mathbf{F}(\alpha)$
lone $v : d \mid \alpha$	$\mathbf{F}$(all $v, v' : d \mid (\alpha$ && $\alpha[v\backslash v'])$ =>$v = v'$) $\overset{✍}{=}$ lone $v{:}\mathbf{T}(d) \mid \mathbf{F}(\alpha)$
one $v : d \mid \alpha$	$\mathbf{F}$((some $v : d \mid \alpha$ && (all $v' : d \mid \alpha[v\backslash v']$=> v = v'))) $\overset{✍}{=}$ lone $v{:}\mathbf{T}(d) \mid \mathbf{F}(\alpha)$
v' is a fresh variable, $\alpha[v\backslash v']$ is the result of replacing each free occurrence of v by v' in α.	

Tab. 3.5: Definition of mapping of Alloy formulas to PDOCFA formulas.

THEOREM 3.2.13: Let $\Sigma \cup \{ \varphi \}$ be a set of Alloy formulas. Then,

$$\Sigma \models \varphi \iff \{ F(\sigma) : \sigma \in \Sigma \} \vdash F(\varphi) \ .$$

Proof. $\Longrightarrow$) If $\{ \mathbf{F}(\sigma) : \sigma \in \Sigma \} \nvdash \mathbf{F}(\varphi)$, then there exists a PDOCFA $\mathfrak{F}$ such that $\mathfrak{F} \models \{ \mathbf{F}(\sigma) : \sigma \in \Sigma \}$ and $\mathfrak{F} \not\models \mathbf{F}(\varphi)$. From Thm. 3.2.10 there exists a proper PDOCFA $\mathfrak{F}'$ isomorphic to $\mathfrak{F}$. Clearly, $\mathfrak{F}' \models \{ \mathbf{F}(\sigma) : \sigma \in \Sigma \}$ and $\mathfrak{F}' \not\models \mathbf{F}(\varphi)$. Then, there must be a relational environment $\mathfrak{p}_{\mathfrak{F}'}$ such that $\mathfrak{A}, \mathfrak{p}_{\mathfrak{F}'} \models_{\mathsf{PDOCFA}} \{ \mathbf{F}(\sigma) : \sigma \in \Sigma \}$ and $\mathfrak{p}_{\mathfrak{F}'} \not\models_{\mathsf{PDOCFA}} \mathbf{F}(\varphi)$. From Lemma A.0.5, there exists an Alloy instance $\mathfrak{a}$ such that $\mathfrak{a} \models \Sigma$ and $\mathfrak{a} \not\models \varphi$. Thus, $\Sigma \not\models \varphi$.
$\Longleftarrow$) If $\Sigma \not\models \varphi$, then there exists an Alloy instance $\mathfrak{a}$ such that $\mathfrak{a} \models \Sigma$ and $\mathfrak{a} \not\models \varphi$. From Lemma A.0.6 there exists a proper PDOCFA $\mathfrak{F}$ compatible with $\mathfrak{a}$. From Lemma A.0.4, $\mathfrak{F}, \mathfrak{p}_{\mathfrak{F}'} \models_{\mathsf{PDOCFA}} \{ \mathbf{F}(\sigma) : \sigma \in \Sigma \}$ and $\mathfrak{p}_{\mathfrak{F}'} \not\models_{\mathsf{PDOCFA}} \mathbf{F}(\varphi)$. Then, $\{ \mathbf{F}(\sigma) : \sigma \in \Sigma \} \nvdash \mathbf{F}(\varphi)$. ■

PDOCFA for Alloy eyes

Taking into account all the notation remarks introduced along the chapter, the reader may note how similar the PDOCFA formulas resulting from the translation process are to the original Alloy formulas. A few minimum changes could make those formulas practically indistinguishable from each other. These changes could be encapsulated in a process of pretty printing to be applied on the resulting PDOCFA formulas. In fact, this technique was used in Dynamite, the tool in which we implement all the ideas presented in this chapter.

The pretty-printing process could do the following: if $\mathsf{iden}_{\mathsf{S}}$ is a PDOCFA constant from an Alloy signature S, it could be swapped for S; every appearance of $\mathsf{iden}_{\mathsf{U}}$ could be replaced by univ or iden, depending on the original Alloy expression; $\mathsf{iden}_{\mathbb{Z}}$ could be pretty-printed as Int; the application of the predicates no, some, lone, and none could be expressed as the juxtaposition of the name and the argument, but keeping aside the parenthesis; finally, all the numeric operations can be rewritten to appear familiar to the Alloy-trained eye.

4. DYNAMITE

4.1 Implementation remarks

Implementing Dynamite required solving two tasks, namely:

1. Providing a shallow embedding into PVS of the PDOCFA theories resulting from the translation of Alloy specifications.
2. The careful design of the interaction between PVS and the Alloy Analyzer required in order to provide the user with the new commands offered by Dynamite.

The proposed solutions are reported in Sections 4.1.1 and 4.1.2, respectively.

4.1.1 Embedding the Alloy Calculus in PVS

Proving Alloy assertions using Dynamite involves generating a PVS specification. Said specification is obtained as a shallow embedding [Gordon (1989)] of the PDOCFA theory resulting from the translation presented in Def. 3.2.12. PDOCFA theories obtained from Alloy models have in common their logical part (operations, their meaning and inference rules) presented in Defs. 3.2.1 and 3.2.9, while they may differ in the extralogical elements (constants, axioms, theorems) that are directly related to the actual Alloy specification used as input of the translation. Accordingly, the resulting PVS specifications are also composed of two parts, one for handling the logical elements of PDOCFA theories, and another for handling the extralogical ones.

Embedding of the Logical Aspects of a PDOCFA theory

In the general part of the specification the following elements are defined:

- A data type (called `Carrier`) representing the set **R** of binary relations from Def. 3.2.1.
- Constants and functions representing the constants and operators from Def. 3.2.1. See table 4.1 for a comprehensive list of the PVS elements representing each proper PDOCFA operator and constant.
- PVS axioms capturing the axioms and inference rules presented in Def. 3.2.9. For instance,

```
RA_1 :AXIOM
FORALL (x, y, z: Carrier): composition(x, composition(y,z))
    =composition(composition(x, y), z)
```

PDOCFA element	PVS embedding
$+$	`sum(x?0,x?1: Carrier) : Carrier`
$\&$	`product(x?0,x?1: Carrier) : Carrier`
$^{-}$	`complement(x?0: Carrier) : Carrier`
$\emptyset$	`zero : Carrier`
$univ$	`one : Carrier`
$.$	`composition(x?0,x?1: Carrier) : Carrier`
$iden$	`one_prime : Carrier`
$\sim$	`converse(x?0: Carrier) : Carrier`
$*$	`RTC(x?0: Carrier) : Carrier`

(Notice that the question mark is a valid character for an identifier in PVS.)

Tab. 4.1: Embedding of PDOCFA symbols and constants from Def. 3.2.1 in PVS.

- Auxiliary constants, operators and predicates, as the ones presented in the previous section (π, ρ, $univ_{\mathsf{U}}$, $\otimes$, in, Point), and a few more intended to facilitate the translation process. All of these elements are defined using the elements mentioned in the preceding items. For example, the binary operation $\otimes$ is defined as follows:

```
Cross(x,y: Carrier): Carrier =
    fork(composition(Pi,x),composition(Rho,y))
```

PVS natively provides a standard sequent calculus (see [Owre et al. (2001b)] for details). The only rule that has to be incorporated is the ω-rule (see Def. 3.2.9). Using the support for natural numbers offered by PVS, this rule is expressed as a PVS axiom.

Embedding of the Extra-logical Aspects of a PDOCFA theory

We present the translation of the extralogical part of the specification in two steps. We focus first on PDOCFA constants (coming from Alloy signatures and fields) and their properties. We subsequently deal with the translation of functions, predicates, axioms and assertions.

When translating an Alloy signature definition it is necessary to introduce a new symbol for the partial universal relation over atoms from that domain, and another new symbol for the partial identity formed with those atoms. For instance, the translation of signature `Agent` yields the PVS definitions[1]:

```
univ_this?Agent: Carrier % the partial universal of Agent atoms
iden_this?Agent: Carrier % the partial identity of Agent atoms
```

[1] Notice that in PVS everything at the right of the `%` symbol is considered a comment, and that `?` is a valid character in identifiers.

In addition, axioms enforcing that these constants have the characteristics mentioned before (being a partial universal relation or a partial identity) must be included. Notice that this part of the translation may generate more axioms depending on the characteristics of the signature being translated (abstract, one sig, extension, etc.).

Restricting quantifiers to range over atoms, as explained in Section 3.2.2, requires adding for each signature a predicate stating that a relation is a point and it is included in the partial universal relation corresponding to that signature. For signature `Agent`, the predicate is[2]:

```
this?Agent(R: Carrier): bool= Point(R) AND Leq(R,univ_this?Agent)
```

When translating field definitions, besides the declaration of the corresponding constant, it is necessary to add appropriate axioms stating the restrictions that the definition imposes on the field. For example, the translation of field `routing` from signature `Domain` leads to the definition of constant `this?Domain?routing` and to the inclusion of the following axiom[3]:

```
this?Domain?routing: AXIOM FORALL (this: (this?Domain)) :
   Leq(this?Domain?routing,
        CartesianProduct(univ_this?Domain,
                 CartesianProduct(
                          Navigation(this,this?Domain?space),
                          Navigation(this,this?Domain?endpoints))))
```

This axiom establishes that `this?Domain?routing` denotes a relation in which, for each tuple $\langle d, i \star g\rangle$, address i is in the space of domain d, and agent g is an endpoint for d.

The translation of predicates, functions, facts and assertions is direct. It is sufficient to translate the formula (or expression) that defines each of these constructs and add the corresponding predicate, function, theorem or axiom to the resulting PVS specification. For example, assertion `BindingPreservesReachability` (shown in Fig. 5.9) is translated to the PVS theorem shown in Fig. 4.1

Pretty-printing of PDOCFA embeddings in PVS

On top of this notation pretty-printing algorithms are applied to PVS formulas occurring during the development of proofs. Therefore, the user only sees Alloy syntax while working within Dynamite. This is one of the important features of Dynamite because it makes it unnecessary for Alloy users to learn another formalism in order to prove the given assertions.

With the aim of illustrate this feature, the Fig. 4.2 shows the sequent obtained after a few rule applications from the beginning of the proof of

[2] `Leq` is the predicate corresponding to the set inclusion operator *in*.

[3] `Navigation` is the operator corresponding to $\bullet$, and `CartesianProduct` simulates the behaviour of Cartesian product between relations of arbitrary arity.

```
BindingPreservesReachability :THEOREM
FORALL (
 d: (this?Domain2),
 d??_: (this?Domain2),
 newBinding: Carrier | Leq(newBinding,
      CartesianProduct(univ_this?Identifier,univ_this?Identifier))
):
 (this?IdentifiersUnused(d,Navigation(newBinding,univ_this?Identifier))
 AND this?AddBinding(d,d??_,newBinding))
    IMPLIES ( FORALL (i: (this?Identifier), g: (this?Agent)) :
      this?ReachableInDomain(d,i,g) IMPLIES
           this?ReachableInDomain(d??_,i,g) )
```

Fig. 4.1: PVS embedding of the PDOCFA theorem corresponding to the Alloy assertion BindingPreservesReachability (see Fig. 5.9).

the theorem `BindingPreservesReachability`, both with and without the application of the pretty-printing procedure.

It is worth noting that not any PDOCFA embedded formula nor expression can be pretty printed as a valid Alloy element. For instance, quantifications with no constraints on the quantified variables (such as the axiom `RA\_1` previously shown) or expressions containing PDOCFA specific operators (such as ∇) can not be pretty-printed. Nevertheless, we have defined carefully the embedding so that all PDOCFA formula or expression resulting from the translation of an Alloy element, may be pretty-printed.

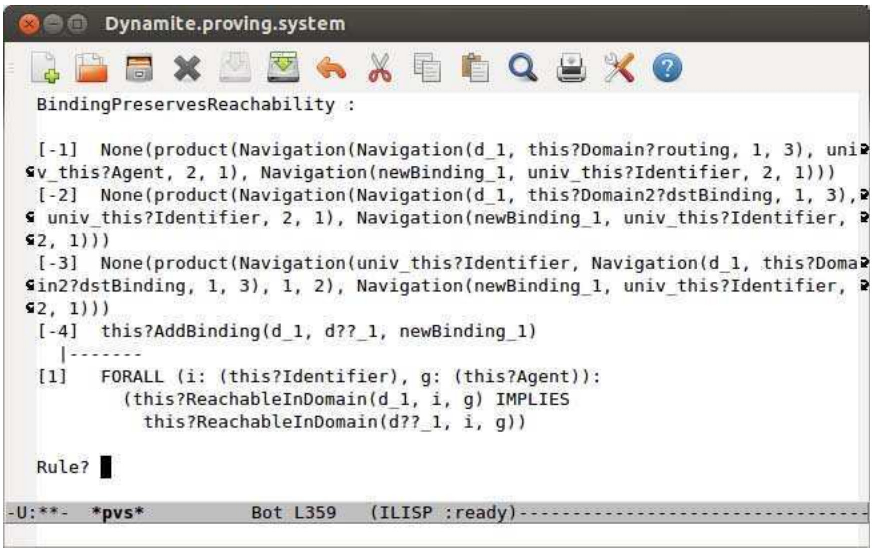
Embedding Alloy integers

The characterization of Alloy integers in PDOCFA presented in Section 3.2.2 can be easily embedded in PVS as a new PVS theory. Such an embedding would be suboptimal, since it would miss all the support provided by PVS for reasoning about integer arithmetic. We will instead use a new PVS theory `fint` (for *finite ints*), parameterized by the bit width. This theory profusely uses theory `int` provided by PVS. For example:

- the bit width (noted as `bitwidth`), is a formal parameter of the theory and has type `posnat` (i.e., positive natural).
- the minimum and the maximum of the interval determined by the bit width are modeled by the integer constants `min_fint` and `max_fint` defined as

$$-\mathtt{exp2}(\mathtt{bitwidth}-1) \quad \text{and} \quad \mathtt{exp2}(\mathtt{bitwidth}-1)-1,$$

  respectively.
- a PVS predicate is defined for delimiting the numbers in this interval:



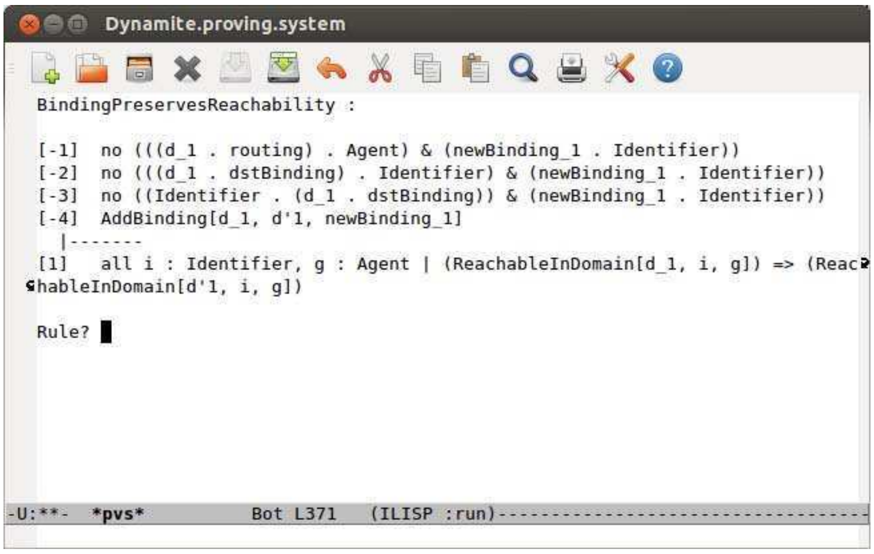
```
BindingPreservesReachability :

[-1]  None(product(Navigation(Navigation(d_1, this?Domain?routing, 1, 3), uni
v_this?Agent, 2, 1), Navigation(newBinding_1, univ_this?Identifier, 2, 1)))
[-2]  None(product(Navigation(Navigation(d_1, this?Domain2?dstBinding, 1, 3),
 univ_this?Identifier, 2, 1), Navigation(newBinding_1, univ_this?Identifier,
2, 1)))
[-3]  None(product(Navigation(univ_this?Identifier, Navigation(d_1, this?Doma
in2?dstBinding, 1, 3), 1, 2), Navigation(newBinding_1, univ_this?Identifier,
2, 1)))
[-4]  this?AddBinding(d_1, d??_1, newBinding_1)
  |-------
[1]   FORALL (i: (this?Identifier), g: (this?Agent)):
        (this?ReachableInDomain(d_1, i, g) IMPLIES
          this?ReachableInDomain(d??_1, i, g))

Rule? 

-U:**-  *pvs*         Bot L359    (ILISP :ready)--------------------------------
```

(a)



```
BindingPreservesReachability :

[-1]  no (((d_1 . routing) . Agent) & (newBinding_1 . Identifier))
[-2]  no (((d_1 . dstBinding) . Identifier) & (newBinding_1 . Identifier))
[-3]  no ((Identifier . (d_1 . dstBinding)) & (newBinding_1 . Identifier))
[-4]  AddBinding[d_1, d'1, newBinding_1]
  |-------
[1]   all i : Identifier, g : Agent | (ReachableInDomain[d_1, i, g]) => (Reac
hableInDomain[d'1, i, g])

Rule? 

-U:**-  *pvs*         Bot L371    (ILISP :run)----------------------------------
```

(b)

Fig. 4.2: Screenshots of Dynamite at one of the first sequents of a proof: *(a)* without pretty-printing, and *(b)* with pretty-printing.

```
inRange_fint(n: int): bool= min_fint<=n and n<=max_fint
```

- this PVS theory includes the definition of a subtype of `int`, called `fint`, that represents the Alloy integers in the interval:

```
fint: TYPE= { n: int | inRange_fint(n) }
```

- all the integer operations supported in Alloy (addition, subtraction, multiplication, integer division and remainder) are modeled as PVS functions on `fint`. For example, addition is defined as

```
add_fint(n1, n2: fint): fint =
   IF inRange_fint(n1+n2) THEN n1+n2
   ELSIF n1+n2 > max_fint
      THEN (min_fint-1)+(n1+n2)-max_fint
      ELSE max_fint+(n1+n2)-(min_fint-1)
   ENDIF
```

As discussed at the end of section 3.2.2, it must be noted that this theory does not support models in which the set of integer atoms is not finite.

4.1.2 Overview of Dynamite's Architecture

The current prototype of the Dynamite Proving System was developed as an extension of PVS. Therefore, we wrote Emacs extensions (for system commands such as those for opening and editing an Alloy specification), Lisp routines that interact with the PVS prover engine (to implement the Dynamite-specific commands, the pretty-printing of the formulas, etc.) and Java code (whose purpose is the translation and validation of formulas, goals and specifications, as well as the postulation of witness candidates for existentially quantified assertions, among others).

A component-and-connector view diagram of Dynamite's architecture showing the interactions between the main components of the system[4] is depicted in Fig. 4.3.

As explained by Owre (2008), the PVS prover engine runs as a subprocess of Emacs, through an ad-hoc ILISP interface [Kaufmann et al. (2002)]. We added the implementation of the Dynamite-specific commands, to be

[4] It is worth noting that, as usual in C&C diagrams, despite being of the same type, not all the client-server connectors showed in the figure are implemented in the same way. For example, the connectors between the "Dynamite Translator" and the Alloy Analyzer are implemented using the API exposed by the latter, while the connectors linking the "Dynamite proof commands processor" and the "Dynamite Translator" are implemented through the OS standard input/output subsystem.

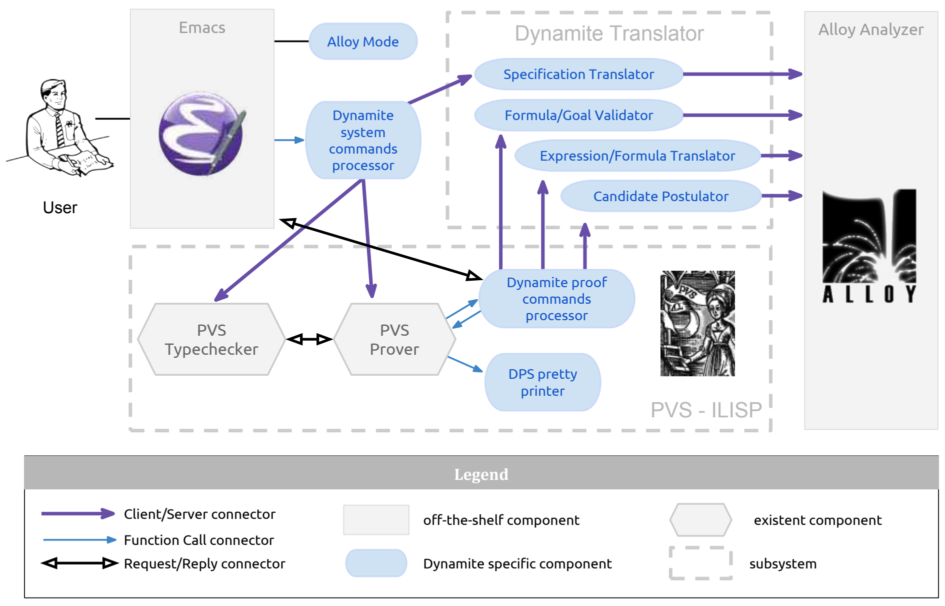


Fig. 4.3: C&C view of the Dynamite architecture.

explained in following sections, to this engine as PVS strategies and rules. “Dynamite proof commands processor” in the diagram. These extensions are conservatively sound with respect to the logic of PVS.

We also modified the PVS function for pretty-printing in order to allow the user to see Alloy formulas in the sequents as long as it can be done. At any point during a proof the user can deactivate and activate the pretty-printer, when the translation back is possible.

The “Dynamite proof commands processor” is responsible for the interaction with the Java processes that, using the Alloy Analyzer, validate formulas and goals during the proof, suggest the elimination of (presumably) unnecessary formulas from the sequent, and postulate expressions that can be used to instantiate existential quantifiers. These Java processes are collectively referred to as “Dynamite Translator”[5].

As Dynamite is an extension of PVS, all the regular PVS proof commands are available to the user. Some of them, such as `case` and `inst`, take formulas or expressions as parameters. When the pretty-printer is activated, the user can write Alloy formulas and expressions as parameters for these commands. The “Dynamite proof commands processor” is also responsible for the translation of those parameters to the corresponding PDOCFA formulas and expressions, via the “Dynamite Translator”.

Dynamite includes Emacs extensions that allow the user to open an Alloy specification, generate the corresponding PDOCFA theories and start,

[5] Besides inaccuracy, the name is maintained for historical reasons.

or redo, the proof of any of its assertions (among other system-level functionalities). All these functions can be accessed through the user menu. Additionally, Dynamite has an Emacs major mode ("Alloy mode") that provides syntax highlighting for the manipulation of Alloy code.

4.2 Features of Dynamite

Proving properties can be seen as a handcraft discipline. It usually requires a high level of training on the methods adopted to develop the proofs, a deep understanding of the concepts formalized in the theory and, most of the time, lots of patience. Even more so if we consider that the person in charge of proving the correctness of the assertions is (many times) not the person who wrote the model.

Automatic verification of proofs is somewhat comparable to the spell checker in text editors. It does not help you compose a text, it just guarantees the absence of syntactic mistakes. To be considered useful, an interactive theorem prover must be capable of helping the user with the proving process.

We already mention in section 2.2.3 a couple of critical points of the proof process where the user might need assistance. Those points heavily rely on specific rules, nevertheless there are other aspects related to the mechanic of the calculus, to the way in which the current sequent is evolving, that also obstruct the developing of the proof.

Dynamite is intended to be more than a proof checker. It offers help in situations such as the previously mentioned. In order to do so, it uses the Alloy Analyzer in different ways that are to be detailed in the following sections. This makes Dynamite much more than syntactic sugar on top of PVS.

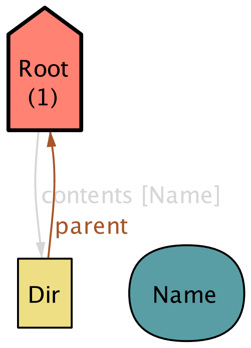
4.2.1 Introduction of Hypotheses and Lemmas

As explained in section 2.2.3, the application of the Cut rule can represent the introduction of a new hypothesis. In PVS the user can introduce a new hypothesis in such a way using the rule case. Still, we may want to go a step further and gain some confidence on the suitability of the introduced hypothesis. Does it actually follow from some of the formulas already in the sequent? It is frustrating to realize, after finishing the proof with the aid of the new hypothesis, that it cannot be discharged, deeming the previous proof effort useless. In order to reduce the risk of introducing inappropriate hypotheses, Dynamite introduces the rule dps-hyp:

$$\frac{\Gamma, \alpha \vdash \Delta \qquad \Gamma' \vdash \alpha, \Delta'}{\Gamma \vdash \Delta}\ \textsf{dps-hyp}(\alpha)$$

where $\Gamma' \subseteq \Gamma$, $\Delta' \subseteq \Delta$ and at least one of $\Gamma \neq \Gamma'$ and $\Delta \neq \Delta'$ holds. The use of rule dps-hyp triggers a call to the Alloy Analyzer in order to

```
 |-------
{1}   lone (contents.o)
```



(a) *(b)*

Fig. 4.4: Example of use of the rule (dps-hyp).

analyze whether sequent $\Gamma' \vdash \alpha, \Delta'$ follows from the model. If a counterexample is found within the provided scopes, it is reported to the user and the hypothesis is removed.

Example 1: Assume we are trying to prove the assertion NoDirAliases presented in Section 2.1.1 (Pg. 11). This assertion, shown below, states that all directory may be contained at most in one directory.

```
assert NoDirAliases { all o: Dir | lone (contents.o) }
```

As the first step of our proof we need to apply the command corresponding to the rule ($\vdash\forall$) (see Figure. 2.3) in order to get rid of the universal quantification. After the application of such command, we would face the sequent depicted in Figure 4.4.a.

Sometimes, stronger properties are easier to prove. We could try to prove that **one** contents.o holds, or that **no** contents.o holds. In other words, if we could prove that contents.o is always empty or it always has only one element, we could complete the proof of the assertion. In order to test this strategy, we can use the Dynamite command `(dps-hyp "one contents.o")`.

Under this configuration, the sets of formulas mentioned in the description of the rule dps-hyp are the following: $\Gamma = \emptyset$, $\Delta = \{$"**lone** contents.o"$\}$, $\alpha =$"**one** contents.o". As its default behaviour, this rule takes $\Gamma' = \Gamma$ and $\Delta' = \emptyset$. So, Dynamite will use the Alloy Analyzer to validate the sequent we should address to prove that this conjecture follows from the original set of hypothesis, i.e. $\emptyset \vdash$"**one** contents.o". The Alloy Analyzer will find the counterexample shown in Figure 4.4.b, where can be seen that the root directory is not included in any other directory, thus invalidating the sequent under analysis. Consequently, the application of the rule is aborted.

Notice that the user has to provide the new hypothesis α and the formulas from Γ and Δ suspected to be the ones which α follows from. It could

$$\dfrac{\dfrac{\Gamma,\neg\alpha \ \vdash\ \Delta}{\Gamma,\neg\alpha \ \vdash\ \alpha,\Delta}\ \mathbf{W}\text{ on }\alpha \qquad \dfrac{\dfrac{}{\Gamma,\alpha \ \vdash\ \alpha,\Delta}\ \text{Ax}}{\Gamma \ \vdash\ \neg\alpha,\alpha,\Delta}\ \vdash\neg}{\Gamma \ \vdash\ \alpha,\Delta}\ (\text{Cut using }\neg\alpha)$$

Fig. 4.5: Strategy to transform the sequent $\Gamma \vdash \alpha, \Delta$ into $\Gamma, \neg\alpha \vdash \Delta$

seem a requirement for an extra effort from the user, but this selection of formulas from Γ and Δ is usually performed at the moment of choosing α even it is generally revealed a few steps after the application of the Cut rule, when the specific proof of $\Gamma' \vdash \alpha, \Delta'$ begins.

4.2.2 Introduction of cases

As the Cut rule in conventional sequent calculus (see Fig. 2.3), the application of the Dynamite dps-case command splits the current branch into two branches, using a provided formula α as a parameter. In one of the branches α appears as a new formula in the antecedent, and it is placed in the consequent in the other branch:

$$\frac{\Gamma, \alpha \vdash \Delta \qquad \Gamma \vdash \Delta, \alpha}{\Gamma \vdash \Delta}\ \textsf{dps-case}(\alpha)\ .$$

Notice that, following the strategy depicted in Fig. 4.5, having α as a proof obligation can be seen as having $\neg\alpha$ as a hypothesis. Thus, any application of the Cut rule can be seen also as the separation in cases of the proof. In the first of them α holds, and in the other it does not.

The Dynamite dps-case command improves over the regular PVS case command, which implements the Cut rule, by using the Alloy Analyzer in order to automatically search for models of the formulas

$$\left(\bigwedge_{\gamma\in\Gamma'} \gamma\right) \wedge \left(\bigwedge_{\delta\in\Delta'} \neg\delta\right) \wedge \alpha, \qquad \text{and} \qquad \left(\bigwedge_{\gamma\in\Gamma''} \gamma\right) \wedge \left(\bigwedge_{\delta\in\Delta''} \neg\delta\right) \wedge \neg\alpha$$

where, $\Gamma' \cup \Gamma'' \subseteq \Gamma$, $\Delta' \cup \Delta'' \subseteq \Delta$ and at least one of $\Gamma \neq \Gamma'$, $\Gamma \neq \Gamma''$, $\Delta \neq \Delta'$ and $\Delta \neq \Delta''$ holds. Notice that if the original sequent is valid, there can not be models for those conjunctions with $\Gamma' = \Gamma'' = \Gamma$ and $\Delta' = \Delta'' = \Delta$. It is expected that the introduction of α (and $\neg\alpha$) may drop formulas from Γ and Δ in each new branch. Again, the selections Γ', Γ'', Δ' and Δ'' from Γ and Δ must be given by the user. By default, Dynamite assumes that $\Gamma = \Gamma' = \Gamma''$ and $\Delta' = \Delta'' = \emptyset$.

The existence of the models guarantees that formula α indeed splits into meaningful cases. If the Alloy Analyzer does not yield a model for any of the formulas, this is reported to the user.

```
{-1}  (a!1 in (i!1 . (*(d!1 . dstBinding!1))))
[-2]  (IdentifiersUnused (d!1, (newBinding!1 . Identifier)))
[-3]  (all i : Identifier| ((i in (newBinding!1 . Identifier)) =>
      (((i in Address) => (i in (d!1 . space!1))) &&
      ((i in AddressPair) => ((i . addr!1) in (d!1 . space!1))))))
[-4]  ((d'!1 . endpoints!1) = (d!1 . endpoints!1))
[-5]  ((d'!1 . space!1) = (d!1 . space!1))
[-6]  ((d'!1 . routing!1) = (d!1 . routing!1))
[-7]  ((d'!1 . dstBinding!1) = ((d!1 . dstBinding!1) + newBinding!1))
[-8]  (g!1 in (a!1 . (d!1 . routing!1)))
 |-------
{1}   (a!1 in (i!1 . (*((d!1 . dstBinding!1) + newBinding!1))))
[2]   (a!1 in (i!1 . (*(d'!1 . dstBinding!1))))
[3]   (a!1 in ((d!1 . dstBinding!1) . Identifier))
```

Fig. 4.6: Example of a sequent obtained after the application of several proof commands.

4.2.3 Hiding sequent formulas

During the development of a proof the amount of formulas in the sequent tends to grow. For example, new hypotheses are introduced when a case splitting is performed. The information expressed in these hypotheses may be useful for closing some branches and useless for some others. Thus, a sequent may contain formulas that are irrelevant to close branches originating in the sequent. For example, after a branch splitting some formulas may no longer be needed for some of the sub-goals, and be necessary to prove others.

In Fig. 4.6 we show an open branch that was obtained during the proof of property `BindingPreservesReachability`, reached after a few applications of proof commands. Notice that, even when the only relevant formulas of the sequent are `1` and `-1`, the other 9 formulas in the sequent obfuscate the job of proving the assertion, turning the sequent very difficult to understand at first glance. This is a situation that occurs quite often. For instance, predicates are typically used to wrap several related concepts which apply in different sub-goals. Using one of those concepts requires us to expand the predicate. Doing this will not only result in the appearance of the desired formula as hypothesis, but the rest of the sub-formulas will also appear as part of the sequent.

To solve this, it is common for interactive theorem provers to provide commands for hiding formulas from a goal, under the assumption that they will not be used. On the other hand, the use of this command also presents a risk. If, by mistake, a relevant formula is hidden, the user will not be able to close the branch. Given a sequent $\Gamma \vdash \Delta$ (with $\Gamma = \{\, \gamma_1, \ldots, \gamma_k \,\}$ and $\Delta = \{\, \delta_1, \ldots, \delta_m \,\}$) result of hiding some formulas, the Dynamite command dps-validate-goal automatically searches for counterexamples of the logical implication between the conjunction of the formulas in the antecedent and the disjunction of the formulas in the consequent:

$$\left(\bigwedge_{1\leq i\leq k} \gamma_i\right) \Rightarrow \left(\bigvee_{1\leq j\leq n} \delta_j\right) . \tag{4.1}$$

In this way, if a counterexample is found, it means that the proof objective cannot be reached because the hypotheses are not sufficient to prove the desired property. If that goal is the result of hiding some formulas from a sequent for which a similar analysis did not return a counterexample, it means that some of the newly hidden formulas were necessary.

4.2.4 Pruning of goals

As we explained in Section 4.2.3, sequents can grow up to a point in which they get very difficult to be understood. A handy and time-saving feature is the use of the Alloy Analyzer that Dynamite does to prune goals.

Let us assume we are proving a sequent $\Gamma \vdash \Delta$ (where $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$ and $\Delta = \{\delta_1, \ldots, \delta_m\}$) from a theory $\Omega = \{\omega_1, \ldots, \omega_n\}$. In order to reduce the proof search space we will try to identify formulas from Γ, Δ and Ω that can be safely removed. Notice that having fewer formulas actually reduces the proof search space. Many proof attempts that could depend on the removed formulas (rules for instantiation, rewriting, or applying strategies) are now avoided. This reduces the number of instantiations of inference rules that the theorem prover has to consider, as well as helps the user stay focused on the relevant parts of the sequent.

Iterative Approach

The first approach we tempted was a rudimentary strategy, implemented by the Algorithm 1. It allows us to determine a set of formulas candidate to be removed. The algorithm attempts to remove each formula φ, and analyzes (using the Alloy Analyzer) whether the sequent obtained *after* formula φ has been removed is valid or not. If the sequent is valid, then φ can be (safely?) removed.

The previous Alloy analysis requires providing a scope for data domains. Therefore, it might be the case that the analysis of formula (4.1) does not return a counterexample, yet the formula indeed has counterexamples in larger scopes. This shows that this technique is not complete, since a necessary formula might be removed (this explains the question mark on "safely" above) and a valid sequent may no longer be derivable. This is not a problem in itself. Hiding formulas based on the user's intuition is not complete

```
iterativeRemove($\Gamma, \Delta, \Omega$)
    for each $\gamma \in \Gamma$ do
        if proves($\Gamma$ - $\gamma$, $\Delta$, $\Omega$) then        /* Procedure ‘proves($\Gamma$, $\Delta$, $\Omega$)’
        checks, using the Alloy Analyzer, whether sequent $\Delta \vdash \Gamma$ holds
        in model $\Omega$.  In Alloy terms, this amounts to checking, having
        as facts the formulas in $\Omega$, the assertion 4.1.  Procedure
        ‘proves’ returns true whenever the Alloy Analyzer does not
        produce a counterexample.  */
            $\Gamma = \Gamma$ - $\gamma$;
        end
    end
    for each $\delta \in \Delta$ do
        if proves($\Gamma$, $\Delta$ - $\delta$, $\Omega$) then
            $\Delta = \Delta$ - $\delta$;
        end
    end
    for each $\omega \in \Omega$ do
        if proves($\Gamma$, $\Delta$, $\Omega$ - $\omega$) then
            $\Omega = \Omega$ - $\omega$;
        end
    end
end
```

Algorithm 1: The iterative pruning algorithm.

either. Since removing formulas does not allow us to prove previously underivable sequents, refining sequents and theories as explained is a sound rule.

UnSAT Core Approach

Some SAT-solvers, such as MiniSat [Eén et al. (2006)] among the ones provided by the Alloy Analyzer, allow one to obtain upon completion of the analysis of an inconsistent propositional theory, an UnSAT-core. An UnSAT-core is a subset of clauses from the original inconsistent theory that is also inconsistent. The UnSAT-core extraction algorithm implemented in MiniSat mostly produces small UnSAT-cores. The Alloy Analyzer converts the propositional UnSAT-core into an Alloy UnSAT-core [Torlak et al. (2008)] (i.e., a subset of the Alloy model that is also inconsistent if the source model was inconsistent). Notice that the Algorithm 1 actually computes an Alloy UnSAT-core. Moreover, it computes a *minimal* Alloy UnSAT-core.

This feature allow us to implement an improved approach for the pruning of sequents. When the Dynamite proof command dps-hide is applied on sequent $\Gamma \vdash \Delta$, the system builds an Alloy model containing the original Alloy model Ω under analysis and an assertion on the validity of formula (4.1). Notice that analyzing with the Alloy Analyzer the newly built model Ω' will not return counterexamples, otherwise sequent $\Gamma \vdash \Delta$ would not be valid. We then request the Alloy Analyzer for an UnSAT-core of Ω'.

```
(-1) (a in (i . ^ ((newBinding + (d.dstBinding)))))
(-2) no (((Identifier . (d.dstBinding)) & (newBinding . Identifier)))
(-3) no ((((d.dstBinding) . Identifier) & (newBinding . Identifier)))
(-4) (not (a in (i . ^ newBinding)))
(-5) (not (a in (i . ^ (d.dstBinding))))
|---
(1) some ai: Identifier |
        ai in i.^newBinding && a in ai.^(d.dstBinding)
```

Fig. 4.7: Sequent with existentially quantified conclusion.

An Alloy UnSAT-core [Torlak et al. (2008)] of Ω' is a subset of Ω' that is also inconsistent (notice that inconsistency is defined up-to the considered scopes). The UnSAT-cores retrieved by the Alloy Analyzer do not need to be minimal, but many times are proper subsets of the premises and consequents of the sequent and theory under analysis. Command dps-hide then hides all those formulas in Ω, Γ and Δ that are not part of the retrieved UnSAT-core.

The relevant formulas in the sequent depicted in Fig. 4.6 (formulas -1 and 1) are automatically identified by applying this command.

4.2.5 Automated Witness Generation

When proving the property BindingPreservesDeterminism, the sequent depicted in Fig. 4.7 is produced. Notice that the formula in the consequent is existentially quantified. According to the proof calculus depicted in Fig. 2.3, in order to prove the sequent we must find a suitable witness (i.e., a term that, when substituted for variable `ai`, makes the resulting sequent provable):

$$\frac{\Gamma \vdash A\{x \leftarrow t\}, \Delta}{\Gamma \vdash (\exists x : A), \Delta} \vdash \exists$$

In order to reduce user intervention, in this section we will present an effective technique that uses the Alloy Analyzer in order to automatically generate witness candidates. Also, we will present several examples where the application of the proposed technique yields the required witnesses.

In Alg. 2 we present the algorithm we use for witness candidate generation. Recall that *environments* are the semantic structures in which Alloy models are evaluated.

In the following paragraphs we will explain the algorithm and argue about its correctness, as well as discuss in what conditions the algorithm may fail to produce a candidate. Afterwards we will describe several experiments we performed.

```
witnessCandidate($\Gamma, \delta$)                    /* $\delta$ has the form some x : $\alpha$[x] */
    $DW \leftarrow \emptyset$; /* $DW$ will store the discarded witnesses found so far */
    $result \leftarrow \emptyset$;
    $\mathcal{E} \leftarrow \mathcal{E}_0$;                  /* $\mathcal{E}_0$ is an environment such that $\mathcal{E}_0 \models \Gamma$ */
    $\alpha' \leftarrow \alpha$;
    while ($result == \emptyset \wedge \mathcal{E}$ != null) do
        if there exists a witness precandidate $t$ in $\mathcal{E}$ then
            if $t$ is a valid witness precandidate then
                $result \leftarrow result \cup \{ t \}$;
            else
                $DW \leftarrow DW \cup \{ t \}$;
                $\mathcal{E} \leftarrow \mathcal{E}_i$;      /* $\mathcal{E}_i$ is the environment in which $t$ failed */
                for each $t' \in shrunkenWitnessesFrom(t)$ do
                /* $shrunkenWitnessesFrom(t)$ are the witnesses with the
                same syntactic structure as $t$ but with every constant $c$
                replaced by a constant denoting a subset of $c$ */
                    if $t'$ is a valid witness precandidate then
                        $result \leftarrow result \cup \{ t' \}$;
                    end
                end
                if ($result == \emptyset$) then
                    if in every environment $t$ contains atom $a$ such that $\alpha'[a]$
                    holds then
                        $\alpha' \leftarrow \alpha'[t \mathbin{\&} x]$;         /* $\alpha'$ has been relativized */
                    else
                        if there is a coverage $C \subseteq DW$ then
                            $result \leftarrow C$;
                        end
                    end
                end
            end
        else
            if $\alpha'$ has been relativized then
                $\alpha' \leftarrow \alpha$;
            else
                $\mathcal{E} \leftarrow$ null;
            end
        end
    end
end
```

Algorithm 2: Algorithm for witness candidate generation.

```
sig A {}
sig B in A {}
one sig x1, x2 in A {}

fact { x1 in B || x2 in B }

assert needsCoverage { some x : A | x in B }
```

Fig. 4.8: Simple Alloy model where coverage is needed to prove the assertion.

The Inputs to the Algorithm (line 1)

Given a sequent $\Gamma \vdash$ `some x : T` $|$ $\alpha(\mathtt{x})$ that has to be proved, the algorithm receives as inputs the set Γ and the formula `some x : T` $|$ $\alpha(\mathtt{x})$ for which the witness must be produced.

Initialization (lines 2–5)

Variable *DW* will store those witness precandidates that are eventually discarded. Variable *result* stores the output of the algorithm, and its content will be discussed in Section 4.2.5. Variable $\mathcal{E}$ is initialized with $\mathcal{E}_0$, any environment in which Γ holds. $\mathcal{E}_0$ is produced by invoking the Alloy Analyzer.

The Output (variable *result*)

Variable *result* returns a *coverage* for the formula under analysis. A coverage is a set of terms $\{ t_1, \dots, t_k \}$ such that the Alloy Analyzer is able to verify the sequent $\Gamma \vdash \alpha(t_1) \mid\mid \cdots \mid\mid \alpha(t_k)$.

The simple Alloy model in Fig. 4.8 shows that coverages are many times necessary. In Alloy notation, signature `B` denotes a subset of `A`, and objects `x1` and `x2` belong to `A` (and since `B` is contained in `A`, also perhaps to `B`). Notice that the fact guarantees that the assertion is indeed valid. Yet no single witness exists. In some environments `x1` will be a witness, and in others the witness will be `x2`. Notice also that:

- x1 **in** B || x2 **in** B holds as per the fact, and
- (x1 **in** B || x2 **in** B) => (**some** x: A | x **in** B) holds.

Therefore, the coverage $\{ \mathtt{x1}, \mathtt{x2} \}$ allows us to prove the existential formula. This reasoning is easily generalized. The proof-schema in Fig. 4.9 shows that a coverage allows us to prove the existentially quantified formula.

Building a Witness Precandidate (line 7)

This is one of the main contributions of Section 4.2.5. The precandidate is built by internalizing Alloy's syntax inside an Alloy model. We will present

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{\Gamma, \alpha(t_1) \ \vdash\ \alpha(t_1)}\,(\mathrm{Ax})
    }{
      \Gamma, \alpha(t_1) \ \vdash\ \mathsf{some}\ x\colon d \ \mid\ \alpha
    }\,(\vdash \exists)
    \qquad
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{}{\Gamma, \alpha(t_n) \ \vdash\ \alpha(t_n)}\,(\mathrm{Ax})
        }{
          \Gamma, \alpha(t_n) \ \vdash\ \mathsf{some}\ x\colon d \ \mid\ \alpha
        }\,(\vdash \exists)
      }{
        \vdots\ \textit{(same applications as below)}\times n
      }
    }{
      \Gamma, \alpha(t_2) \mid\mid \ldots \mid\mid \alpha(t_n) \ \vdash\ \mathsf{some}\ x\colon d \ \mid\ \alpha
    }\,(\mid\mid \vdash)
  }{
    \Gamma, \alpha(t_1) \mid\mid \ldots \mid\mid \alpha(t_n) \ \vdash\ \mathsf{some}\ x\colon d \ \mid\ \alpha
  }\,(\mid\mid \vdash)
  \qquad
  \dfrac{
    \dfrac{}{\Gamma \ \vdash\ \alpha(t_1) \mid\mid \ldots \mid\mid \alpha(t_n)}\,(\mathrm{Coverage})
  }{
    \Gamma \ \vdash\ \alpha(t_1) \mid\mid \ldots \mid\mid \alpha(t_n), \mathsf{some}\ x\colon d \ \mid\ \alpha
  }\,(\mathbf{W})
}{
  \Gamma \ \vdash\ \mathsf{some}\ x\colon d \ \mid\ \alpha
}\,(\mathrm{Cut})
$$

Fig. 4.9: Use of coverage $\{ t_1, \ldots, t_n \}$ for proving an existential formula.

```
sig A { }
one sig cA extends A { }
sig B { f : A }

assert existentialAssert { some x : B | α(x) }
```

Fig. 4.10: A Sample Alloy Model.

the technique by means of a simple running example. Let us consider the Alloy model presented in Fig. 4.10. The model is instrumented with appropriate signatures, functions and predicates. In Fig. 4.11 we present a fragment of the resulting Alloy model.

The instrumented model introduces new signatures that model syntactic internalizations of the source model signatures and fields, as well as of the relational operators (in Fig. 4.11 we only include the union of unary relations and the intersection of binary relations). The first three lines are exactly the same as the original model. They are the semantic base. Then, the abstract signature Term represents the syntactic terms. Its children UnaryTerm and BinaryTerm represent terms denoting sets and terms denoting binary relations, respectively. Fields unaryValue and binaryValue relate each term with its intended meaning. For example, we define a signature A_Syntax representing the syntactic representation of the signature A in the original module. So, A_Syntax is a one signature, because there is only one symbol for A, and A_Syntax and A are related with each other trough the field unaryValue. In the case of operators, see for example the signature UnarySum. It represents the union between two terms denoting sets. The signature axiom takes care of the meaning of the term, restricting the field unaryValue to be the union of the meaning of the sub terms, and takes care of its complexity, stating that it is equal to the sum of the complexity of both sub terms plus one.

We also include a number of facts that preclude redundant instances. For example, fact `UnarySumOperand0IsNotUniv` states that the first operand in a sum cannot be the universal relation (after all, the result of the union would be the set `univ`). Several other properties of this kind are included.

Finally, using the Alloy Analyzer we look for an environment that satisfies formula `witnessSearch`. The environment allows us to retrieve a term `t` that denotes a nonempty set in which all objects satisfy formula `alpha`. This is the witness precandidate. In order to prevent the analysis from returning previously discarded terms, the model includes a fact that is iteratively enriched in order to prevent previously discarded terms from being produced.

```
sig A {}
one sig cA extends A {}
sig B { f : A }

abstract sig Term {
  complexity : Int
}
abstract sig UnaryTerm extends Term {
  unaryValue : set univ
}
abstract sig BinaryTerm extends Term {
  binaryValue : univ −> univ
}
one sig UnivSyntax extends UnaryTerm{}{ unaryValue = univ }
one sig A_Syntax extends UnaryTerm{}{ unaryValue = A }
one sig cA_Syntax extends UnaryTerm{}{ unaryValue = cA }
one sig B_Syntax extends UnaryTerm{}{ unaryValue = B }
one sig f_Syntax extends BinaryTerm{}{ binaryValue = f }

fact sigComplexity{
  A_Syntax.complexity=1 and cA_Syntax.complexity=1 and
  B_Syntax.complexity=1 and f_Syntax.complexity=2}

sig UnarySum extends UnaryTerm {
  operand0, operand1 : UnaryTerm
}{ complexity = operand0.complexty +operand1.complexity +1
   unaryValue = operand0.unaryValue +operand1.unaryValue }

sig BinaryIntersection extends BinaryTerm {
  operand0, operand1 : BinaryTerm
}{ complexity = operand0.complexty +operand1.complexity +1
   binaryValue = operand0.binaryValue & operand1.binaryValue }

fact UnarySumOperand0IsNotUniv {
  (all t : UnarySum | t .operand0 !in UnivSyntax)) }

run witnessSearch {
  some t : UnaryTerm |
    some t.unaryValue and all v : t.unaryValue | α(v) }
```

Fig. 4.11: Fragment of Alloy model with internalized Alloy syntax.

Validating a Witness Precandidate (lines 8–9)

A witness precandidate `t` is valid in an environment. In contrast, a witness candidate must be valid in *all* environments. In order to analyze whether `t` can be considered as a witness candidate, we modify the Alloy model from Fig. 4.10 by replacing assert `existentialAssert` by the following:

```
assert witnessPrecandidateValidation { some t and all v : t | α(v) }
```

The assertion requires `t` to denote a nonempty set in which all the elements satisfy formula `alpha` *in all environments*. If no counterexamples are produced by the Alloy Analyzer, term `t` is stored in the algorithm output variable *result* and promoted to witness candidate.

Witness Precandidate Failure by Over-Approximation (lines 13–20)

Let us assume that witness precandidate `t` fails in environment $\mathcal{E}_i$. According to Section 4.2.5, this implies that either `t` is empty in $\mathcal{E}_i$ or some value from the set denoted by term `t` does not satisfy α. The second condition may hold even if `t` provides at least one value that satisfies α in each environment. In this case we say that term `t` *over-approximates* a witness candidate. In order to avoid this over approximation we will use two techniques:

1. Recalling that Alloy signatures are constants that may appear in term t, term t may fail to be a witness candidate because it includes a signature that is larger than necessary. For instance, in [Zave (2006)] we have as part of the signature hierarchy

   sig Domain3 **extends sig** Domain2 **extends sig** Domain .

   Semantically, the sets denoted by the signatures satisfy

   $$Domain3 \subseteq Domain2 \subseteq Domain .$$

   Therefore, given a term t of the form Domain.routing that fails to be candidate, Dynamite explores whether the terms Domain2.routing or Domain3.routing are indeed candidates. The technique is applied in lines 13–17.

2. The witness candidate could then be the intersection of `t` with another term. We explore this possibility in lines 18–20.

Over-approximation can be checked with the aid of the Alloy Analyzer by checking the assertion

```
assert overApproximation { some v : t | α(v) }
```

Witness Precandidate Failure by Under-Approximation (lines 22–23)

As explained in Section 4.2.5, term `t` may fail to be a witness precandidate because in some environment e it denotes a set that contains an object that does not satisfy α. Yet there might be an already discarded witness `t'` that satisfies α in environment e. We then explore if there is a subset of the discarded witnesses that jointly with `t` form a coverage. If a coverage exists, it is returned in variable *result*. A coverage is determined with the aid of the Alloy Analyzer by checking assertion

```
assert underApproximation { all v:  univ |
  (v in t1 => alpha(v)) || ··· || (v in tn => α(v))}
```

Lack of Witness Precandidates (lines 29–33)

If a new witness precandidate is not found in the selected environment, it may be due to, essentially, four reasons:

1. the existentially quantified formula is not true within the prescribed scopes,
2. the bound on the complexity of terms considered is smaller than required (new witness precandidates might be found if the complexity bound were increased),
3. formula α' is relativized, and therefore part of the available complexity is spent on the relativization term, and not enough complexity is left to build a new precandidate,
4. the included strategies fail to produce a witness.

In the first case the lack of precandidates may be due to the exhaustion of all the witness precandidates, and the algorithm should terminate without returning a precandidate. Similarly, in the second case the algorithm should be run again but with an increase in the complexity bound. The third case will occur when formula α' is relativized. Therefore, we will remove the relativization in order to enable the search for further witness precandidates. In the fourth case, new strategies should be added to the algorithm.

Termination and Correctness

Termination is guaranteed because in each loop iteration a new witness precandidate must be generated; due to the bound on term complexity only finitely many terms can be generated. Correctness, understood as producing a witness regardless of the analysis scopes, cannot be achieved due to the undecidability of classical first-order logic. Therefore, the algorithm may produce witness candidates that are not suitable for finishing the proof. The effectiveness of the algorithm is evaluated experimentally below.

Limitations

This technique heavily relies on the model-finding ability of the Alloy Analyzer. Consequently, it shares the same limitations. When the search for a candidate begins, limits on the search space must are established by fixing a scope. Unlike standard Alloy models where the scope only constrains the explored semantic environments, Alloy models used for witness generation internalize Alloy's syntax as well. Therefore, besides providing scopes for data domains, we must also provide scopes for syntactic domains (maximum amount of occurrences of each operator symbol in the candidates, for example). If the limits imposed to the search are too restrictive, no admissible candidate will be generated. Otherwise, if the limits are too lax, the search can take too much time or lead to an *out of memory* exception.

5. EVALUATION

In order to evaluate Dynamite, we proceeded in two stages. First, we took an Alloy model of addressing in the context of computer networks, by Zave (2005), and prove five of its assertions using Dynamite without the helping features. We call Dynamite 1.0 this version of the theorem prover. Then, we evaluated the features detailed in Sec. 4.2 working on the proofs of assertions from some toy models, a specification of the Mondex electronic purse system [Ramananandro (2008)], and an Alloy model of binding in network domains [Zave (2006)].

This chapter shows the results we found in these evaluations and some remarks about it. In the next section we explain the test of Dynamite 1.0. The evaluation of the helping features is detailed in Sec. 5.2. Both Sections include a description of the models by Zave (2005, 2006), since the most of the proofs we performed were taken from those works.

5.1 Evaluating Dynamite as a plain prover for Alloy

Besides the features explained in Sec. 4.2, we wanted to know if Dynamite, as a regular sequent calculus theorem prover for Alloy assertions, was a usable tool and how many of the lemmas of a significant case study could be proved without having to switch to PDOCFA language. We choose to work on the model detailed below because it is a formal model interesting by itself, not especially made for this evaluation. In this way, we try to reduce the possibility of take a biased case study[1].

5.1.1 Case Study: Addressing for Interoperating Networks

Zave (2005) presented a formal model of addressing for interoperating networks. These networks connect *agents*, which might be hardware devices or other software systems. Agents can be divided between users of the networking infrastructure, called *client agents*, or being part of the infrastructure, called *server agents.* Agents can use resources from domains, to which they must be *attached.* In order to be able to reach clients from domains, pairs $\langle address, domain\rangle$ are assigned to clients. Different sorts of objects can be distinguished in the previous description.

Domains can create persistent connections between agents. Such connections are called *hops.* Besides the domain that created it, a hop contains information about the initiator and acceptor agents taking part in the connection, and also source and target addresses. A fact forces these addresses

[1] These results were previously reported in [Frias et al. (2007)].

to correspond to the agents (according to the domain map).

Multi-hop connections are enabled by the servers. These connections are called *links*. Links contain information about the server enabling the connection, and about the connected hops. The reflexive-transitive closure of the accessibility relation determined by links is kept by an object "Connections", which also keeps the relation established by the links.

Interoperation is considered a *feature* of networks. Features are installed in domains and have a set of servers from that domain that implement them. Among the facts related to features, we find that each feature has at least one server, and that each server implements exactly one feature. Interoperation features are then characterized by signatures Feature and InteropFeature in Fig. 5.1, where a simplified[2] description of the signatures that formalizes the model detailed here is presented.

An interoperation feature translates addresses (by means of the relation interTrans) between different domains. This is necessary because whenever a client from the feature's domain wishes to connect to a client attached to a different domain, it must have a target address it can use in its own domain space. Of course, the target client must have an address in each domain from which it is to be reached. Different facts are introduced in order to fully understand an interoperation feature behavior, and the following assertions are singled out:

- ConnectedIsEquivalence, asserting that field connected is indeed an equivalence relation (reflexive, symmetric and transitive).
- UnidirectionalChains, asserting that two hops are connected through a link in an ordered manner (one can be identified as *initiator* and the other one as *acceptor*).
- Reachability, asserting that whenever a client c publishes an address a in a domain d ($\langle a, d\rangle \in c.\mathsf{knownAt}$), clients c' from domain d can effectively connect to c.
- Returnability, asserting that if a client c accepted a connection from another client c', then a hop from c can be extended to a complete connection to client c'.

The Alloy description of the previous assertions is given in Fig. 5.2.

5.1.2 Evaluation

We proved the properties in Fig. 5.2 using Dynamite. Fig. 5.3 illustrate the pretty printer feature. In the left side can be seen the initial node of the proof of the assertion Returnability as the plain PVS embedding of the corresponding

[2] The complete model can be obtained from `http://www2.research.att.com/~pamela/svcrte.html`.

```
sig Agent{ attachments: set Domain }
sig Server extends Agent { }
sig Client extends Agent { knownAt: Address -> Domain }
sig Domain{ space: set Address, map: space -> Agent }

sig Hop {
  domain: Domain,
  initiator, acceptor: Agent,
  source, target: Address
}

abstract sig End { }
one sig Init, Accept extends End { }

sig Link {
  agent: Server,
  oneHop,anotherHop: Hop,
  oneEnd,anotherEnd: End
}{
  oneHop != anotherHop
  oneEnd in Init => agent=oneHop.initiator
  oneEnd in Accept => agent=oneHop.acceptor
  anotherEnd in Init => agent=anotherHop.initiator
  anotherEnd in Accept => agent=anotherHop.acceptor
}

one sig Connections {
  atomConnected, connected: Hop -> Hop
}

abstract sig Feature {
  domain: Domain,
  servers: set Server
}
sig InteropFeature extends Feature {
  toDomain: Domain,
  exported, imported, remote, local: set Address,
  interTrans: exported some -> some imported
}{
  domain != toDomain &&
  exported in domain.space && remote in exported
  imported in toDomain.space && local in imported
  remote.interTrans = local
}
```

Fig. 5.1: Simplified model for addresses, agents and domains.

```
assert ConnectedIsEquivalence { all c: Connections |
    (all h: Hop | h in h.(c.connected) ) &&
    (all h1,h2: Hop | h1 in h2.(c.connected) => h2 in h1.(c.connected)) &&
    (all h1,h2,h3: Hop | h2 in h1.(c.connected) && h3 in h2.(c.connected)
        => h3 in h1.(c.connected) ) }

assert UnidirectionalChains { all l: Link |
        (l.agent = l.oneHop.acceptor && l.agent = l.anotherHop.initiator) ||
        (l.agent = l.oneHop.initiator && l.agent = l.anotherHop.acceptor) }

assert Reachability {
    all c: Connections, g1, g2: Client, h: Hop, a: Address, d: Domain |
        g1 = h.initiator && d = h.domain && a = h.target &&
        (a->d) in g2.knownAt
    => (some h2: Hop | g2 = h2.acceptor && (h->h2) in c.connected) }

assert Returnability {
  all c: Connections, g1, g2: Client, h1, h2, h3: Hop |
    h1.initiator = g1 && h2.acceptor = g2 && (h1->h2) in c.connected &&
    h3.initiator = g2 && h3.domain = h2.domain && h3.target = h2.source
   => (some h4: Hop | h4.acceptor = g1 && (h3->h4) in c.connected) }
```

Fig. 5.2: Assertions by Zave (2005).

PDOCFA theorem. In the right side, the pretty printed version of the same sequent is depicted. Notice that the pretty printed version closely resembles the Alloy definition. Furthermore, it can even be compiled with the Alloy Analyzer.

We have shown that it is possible to make proofs within the presented calculus with the aid of Dynamite. We now present some empirical data that will allow readers to have a better understanding of the usability of the tool.

The proofs were carried out by a student who had just graduated, and had no previous experience neither with Alloy, nor with PVS. The estimated time he spent in order to master the proof process is the following:

- 5 days to learn Alloy's syntax and semantics.
- 15 days to learn PVS, including the understanding of the proof rules.
- 40 days to prove all the assertions contained in the Alloy model.
- 15 days to prove the nontrivial required lemmas about PDOCFAs. These lemmas can be considered as *infrastructure* lemmas, that will be reused in future proofs.

```
FAL_Returnability :

  |-------
{1}   FORALL (hDm: (hop_domain), fDm: (feature_domain),
        tDm: (toDomain), tar: (target), rem: (remote),
        aCn: (atomConnected), con: (connected), oHp: (oneHop),
        aHp: (anotherHop), rBy: (reachedBy), map: (map),
        acc: (acceptor), srv: (servers), exp: (exported),
        imp: (imported), loc: (local), iTr: (interTrans),
        spc: (space), agn: (agent), oEd: (oneEnd),
        aEd: (anotherEnd), ini: (initiator),
        att: (attachments), src: (source)):
      FORALL (g1, g2: (Client), h1, h2, h3: (Hop)):
        Navigation_2(h1, ini)=g1 AND Navigation_2(h2, acc)=g2
        AND Leq(composition(composition(h1, one), h2),
                Navigation(cConnections, con))
        AND Navigation_2(h3, ini)=g2
        AND Navigation_2(h3, hDm)=Navigation_2(h2, hDm)
        AND Navigation_2(h3, tar)=Navigation_2(h2, src)
        IMPLIES
        (EXISTS (h4: (Hop)):
           Navigation_2(h4, acc)=g1 AND
           Leq(composition(composition(h3, one), h4),
                Navigation(cConnections, con)))
```

(a)

```
FAL_Returnability :

  |-------
{1}   all g1,g2: Client, h1,h2,h3: Hop |
    (h1.ini)=g1 AND (h2.acc)=g2 AND
    (h1->h2) in (cConnections.con) AND
    (h3.ini)=g2 AND (h3.hDm)=(h2.hDm) AND
    (h3.tar)=(h2.src)
   IMPLIES
    (some h4: Hop |
       (h4.acc)=g1 AND
       (h3->h4) in (cConnections.con))
```

(b)

Fig. 5.3: The initial node of the proof of the assertion Returnability *(a)* without pretty printing process being applied, and *(b)* after the application of the pretty printing routine.

Recall that relations of rank greater than 2 are encoded as binary ones. Therefore, it may be necessary to prove properties that deal with the representation. These are the only proofs that would not be completely natural to an Alloy user. The proof of all the assertions in the model comprises 285 lemmas, of which only 12 use this kind of properties. Moreover, the 12 lemmas actually use 8 different properties of the representation because 3 properties are used at least twice.

Table 5.1 shows some numerical information about the proofs of the specific assertions. The rightmost column shows the time in days the user spent in the proof of each assertion and the lemmas used in it. Notice that the sum of the total of lemmas amounts to 365. Therefore, $365 - 285 = 80$ lemmas were re-used in the proof of different assertions. For the count of the time effort, the time spent in proving each re-used lemma only is taken into account once in all the table.

Assertion	Total Lemmas	Model Lemmas	Algebra Lemmas	Time (days)
ConnectedIsEquivalence	79	4	75	10
UnidirectionalChains	52	28	24	5
Reachability	121	62	59	23
Returnability	113	66	47	17

Tab. 5.1: Distribution of the workload.

5.2 Evaluating characteristic features of Dynamite

The features detailed in Sec. 4.2 were evaluated mainly using assertions for an Alloy model by Zave (2006). We begin this section with a explanation of that model. Then, the evaluation of the features are presented in the following order: Sec. 5.2.2 presents the evaluation of the sequent pruning techniques, Sec. 5.2.3 detail the results of testing of the witness generation feature. Sec. 5.2.4 outlines some remarks about the use of the Alloy Analyzer to validate lemmas, new hypothesis and sequents, at a given point of the proof.

5.2.1 A Running Example: Binding in Network Domains

The model, presented in Zave (2006), deals with the formal definition of a mechanism for binding of identifiers in the delivery of messages in computer networks. Thus, the model is not a specification of an isolated software or hardware artifact, but rather the specification of network services whose implementation may involve several software and hardware *agents*. The model describes how communicating agent identifiers are bound so that the messages reach their correct destination. Properties about the possibility of reaching an agent, determinism in the delivery of messages, existence of cycles in the routing of messages and the possibility of constructing a return path for a message are formally specified in the model. In particular, the model studies how these properties are affected by the addition of new bindings between identifiers.

When an agent wants to send a message to another agent a *communication* is established. That communication may involve intermediary agents that just forward the message in its way to its destination. The original sender of the message and its intended final receiver are called the *endpoints* of the communication. Endpoints are organized into *domains*. Each domain has its own set of endpoints, and uses identifiers to recognize them. Identifiers are called *addresses*. Additionally, a domain keeps track of how agents are identified by particular addresses. *Paths* describe connections from a *generator* agent to an *absorber* agent assuming the generator can be recognized by address *source* and the absorber by address *dest*. A simplified version of these concepts in Alloy takes the form depicted by Fig. 5.4.

A domain *supports* a path if the connections described by the path are consistent with the domain. The predicate whose declaration is shown in Fig. 5.5 characterizes when a domain supports a path.

When a message has to be send to an endpoint, the identifier used by the initiator to indicate the destination must be bound to the identifier used by the domain to locate the receiver. This binding is done in three ways. The simplest one is when the initiator is responsible for performing the binding. The message sent has as destination the actual identifier of the receiver. The

```
sig Agent { }

abstract sig Identifier { }

sig Address extends Identifier { }

sig Domain { endpoints: set Agent,
    space: set Address,
    routing: space −> endpoints }

sig Path { source, dest: Address,
    generator, absorber: Agent }
```

Fig. 5.4: Main signatures in the model by Zave (2006).

second scenario occurs when the message sent by the initiator is delivered to an agent that is not the intended receiver. This agent, called *handler*, looks up the corresponding binding, updates the destination address and forwards the message. The third one is basically the same as the second one, but the original destination identifier is composed of two parts which are used by the handler to locate the next agent in the forwarding chain.

These communication patterns show the need for some distinction in the identifiers used in the model, which are stated in the model by extending the previous signatures as depicted in Fig. 5.6. Besides addresses, there will be unrestricted identifiers called *names*, and complex identifiers used in the third kind of communications. Thus, signature Identifier is extended by signatures Name (modeling unrestricted identifiers) and AddressPair (for compound identifiers). Two fields in the latter, addr and name, are defined to formalize the structure of complex identifiers. The possible bindings in each domain are specified by a ternary relation

$$\mathsf{dstBinding} \subseteq \mathsf{Domain} \times \mathsf{Identifier} \times \mathsf{Identifier}\ .$$

We introduce dstBinding (destination binding) by extending the signature Domain, and constraining its meaning with a signature axiom. Paths are also extended in order to include a new field origDst representing the identifier originally given as destination.

```
pred DomainSupportsPath [d: Domain, p: Path] {
   p.source in (d.routing).(p.generator) and
   p.absorber in (p.dest).(d.routing)
}
```

Fig. 5.5: Declaration of predicate DomainSupportsPath.

```
sig Name extends Identifier { }
sig AddressPair extends Identifier { addr: Address, name: Name }

sig Domain2 extends Domain { dstBinding: Identifier −> Identifier } {
 all i: Identifier | i in dstBinding.Identifier implies
   ( (i in Address implies i in space) and
     (i in AddressPair implies i.addr in space) )}

sig Path2 extends Path { origDst: Identifier }
```

Fig. 5.6: Extension to signatures in Fig. 5.4.

```
pred AddBinding[d,d': Domain2, newBinding: Identifier −> Identifier] {
−− Precondition: the new bindings can be applied in the domain.
   all i: Identifier | i in newBinding.Identifier implies
   ( (i in Address implies i in d.space) and
      (i in AddressPair implies i.addr in d.space)
   ) and
−− Postconditions:
   d'.endpoints = d.endpoints and
   d'.space = d.space and
   d'.routing = d.routing and
   d'.dstBinding = d.dstBinding +newBinding
}
```

Fig. 5.7: Alloy predicate modeling the addition of a new binding (newBinding) to a domain (d). The variable d' represents the resulting domain, that contains newBinding and all the bindings from d if the new binding could be added to d according to the preconditions stated in the predicate.

A predicate AddBinding states how a domain is affected by the addition of new bindings (Fig. 5.7).

In the context of this model, an agent g is considered "*reachable* in a domain d from an identifier i" if:

- i is connected to an address a in the reflexive–transitive closure of the binary relation formed by all the bindings corresponding to d,
- a cannot be bound to another identifier in d, and
- in domain d, a can route messages to g.

In Alloy, reachability is modeled by the predicate shown in Fig. 5.8.

Figure 5.9 presents assertion BindingPreservesReachability. This assertion states that if an agent is reachable in a domain d, it is also reachable in the domain resulting from adding a new binding to d, provided that the newly

```
pred ReachableInDomain [d: Domain2, i: Identifier, g: Agent] {
   some a: Address |
      a in i.*(d.dstBinding) and
      a !in (d.dstBinding).Identifier and
      g in a.(d.routing) }
```

Fig. 5.8: Predicate stating the reachability of an identifier in a domain.

```
assert BindingPreservesReachability {
   all d,d': Domain, newBinding: Identifier−>Identifier |
      IdentifiersUnused[d,newBinding.Identifier] and
      AddBinding[d,d',newBinding]
        implies (all i: Identifier, g: Agent |
                    ReachableInDomain[d,i,g]
                         implies ReachableInDomain[d',i,g]) }

pred IdentifiersUnused [d: Domain2, new: Identifier ] {
   no ( (d.routing).Agent & new ) and
   no ( (d.dstBinding).Identifier & new ) and
   no ( Identifier.(d.dstBinding) & new ) }
```

Fig. 5.9: A nontrivial assertion: BindingPreservesReachability.

bound identifiers are not used in d. This latter condition is formalized by the predicate IdentifiersUnused.

A domain is called *deterministic* if each identifier is associated to at most one agent. Assertion BindingPreservesDeterminism states that

> *whenever a new binding for an unused identifier is added to a deterministic domain, it remains deterministic.*

A domain is considered *non-looping* if the transitive closure of the bindings for that domain has no cycles. Assertion BindingPreservesNonlooping then states that

> *the addition of a new binding to a non-looping domain keeps this condition as long as the transitive closure of the new binding does not have cycles.*

Another desirable property of a network is the capability to send a message to the sender of a previously received message. This is called *returnability*. A domain in which it is possible to return the received messages is called a *returnable* domain. In order to write conditions ensuring returnability of a domain, it is necessary to study how the source identifiers can be modified by the handlers that forward the message, because the final source

	NoRec	Iter	UnSAT
Proofs' length	969	597	573
Average # of formulas per sequent	5.89	6.01	6.20
Occurrences of formulas in proofs (no theories)	5706	3590	3215
Average # of formulas in sequents or theories	34.89	35.01	7.02
Occurrences of formulas in proofs (with theories)	33807	20903	4023
# SAT-solver calls	N/A	770	69
# times UnSAT-core missed formulas	N/A	N/A	1
# times UnSAT-core avoided detour	N/A	N/A	2

Tab. 5.2: Measures of attributes for the employed techniques (N/A = not applicable).

identifier is used by the receiver as the destination of the return message. An assertion StructureSufficientForReturnability is also modeled in [Zave (2006)].

Zave (2006) used the Alloy Analyzer to analyze the model and concluded that the previously presented assertions hold for Alloy domains containing at most 2 network domains and 4 elements in each set (such as identifiers, agents, etc). Using Dynamite we proved that all these assertions hold independently of the maximum amount of elements in each set.

5.2.2 Pruning of goals

In this section we present some experimental results we have obtained while applying both pruning techniques explained in Sec. 4.2.4[3]. We begin by presenting some statistics about the model being verified. We have verified the model in three different ways, namely: *(1)* without using any technique for refining the sequents and theories. (noted as NoRec –for *no recommendation*– in Table 5.2); *(2)* using the iterative algorithm in order to refine sequents (noted as Iter –for *iterative recommendation*–); and *(3)* using the UnSAT-core extraction technique (UnSAT). Notice that the way NoRec *(1)* corresponds to verification using Dynamite 1.0.

In Table 5.2 we measure for each technique:

- Length of proofs (measured as the number or rule applications).
- Average number of formulas per sequent.
- Sum of occurrences of formulas in proof sequents.
- At each proof step the user, and some PVS commands, must consider sentences from the current sequent as well as the sentences from the underlying theory. We then measure the average number of such formulas over the different proof steps.

[3] These results were previously reported in [Moscato et al. (2010)].

- Sum (over the proof steps) of occurrences of formulas in proof sequents or from the underlying theories.
- Number of SAT-solver calls for the iterative and the UnSAT-core-based techniques.
- Number of times the UnSAT-core obtained missed a formula necessary for closing a proof branch.
- Number of times the UnSAT-core allowed us to remove formulas that were used in the original proof because of an unnecessary detour.

In order to focus on the most relevant data we are ignoring proof steps where we prove Type Check Constraints (TCCs), which in general can be proved in a direct way. Also, we only applied the techniques (either the iterative or the UnSAT-core-based) on 69 proof steps where it was considered relevant to apply the rules. Systematic application of the iterative technique (for instance each time a new proof goal was presented by PVS) would have required in the order of 25000 calls to the SAT-solver. As a general heuristic, we set the scope for all domains (in the calls to the Alloy Analyzer) to 3.

Notice that proofs carried out using any of the techniques for sequent and/or theory refinement are about 40% shorter than the original proof.

In the original proof, as a means to cope with sequents' complexity, formulas that were presumed unnecessary were systematically hidden. While the average number of formulas per sequent is smaller for the original proof, having half the proof steps shows that the automated techniques are better focused on the more complex parts of proofs. This is supported by the analysis of the total number of formulas that occur in sequents. The UnSAT-core-based technique uses 56% of the formulas used in the original proof, while the iterative technique uses 63% of the formulas.

Since the underlying theory in the case-study has 29 formulas, the overhead in applying the iterative technique to formulas in the theory was too high. Therefore, the iterative technique was only applied to formulas occurring in the sequents being verified along a proof (we believe this will be the case most times). On the other hand, the UnSAT-core extraction receives the current sequent plus the underlying theory, and automatically refines *also* the theory. This explains the big difference between the average number of formulas involved in proofs (both in sequents and in the supporting theory) using the iterative technique and the UnSAT-core-based technique. Notice that this implies that in each proof step the user, and some PVS commands, had to consider significantly fewer formulas in order to suggest further proof steps.

Since proofs are shorter and each sequent contains possibly fewer formulas, the total number of formulas occurring in proofs using UnSAT-cores reduces from the original proofs in about 88% (recall that hiding was also used in the original proofs but not in an automated way, and that formulas from the underlying theory were not hidden). For the iterative technique, the number of formulas reduces in about 40%.

While using UnSAT-cores required only 69 calls to the SAT-solver, the corresponding proof steps using the iterative algorithm required 770 calls to the SAT-solver (without making calls for formulas occurring in the underlying theory). Thus, the UnSAT-core-based technique requires under 10% of the calls required by the iterative technique.

Often during the original proof necessary formulas were incorrectly hidden. We do not have precise records of the number of times this happened because those erroneous proof steps (which at the time were not considered important) were most times undone. We only kept track of 9 cases where the `reveal` command was used in order to exhibit a previously hidden formula, but these were just a few of the cases. It is worth comparing with the single case where the UnSAT-core-based technique missed a formula. This missed formula is recovered if instead of using a scope of 3 in calls to the Alloy Analyzer, scope 5 is used.

Recalling that we have proved 5 Alloy assertions, the proofs of assertions BindingPreservesDeterminism and BindingPreservesNonLooping required fewer formulas during the proof based on UnSAT-cores. This shows that the original proof used unnecessary formulas that were removed using rule `dps-hide`.

A more qualitative analysis of the techniques allows us to conclude that refining sequents and theories using UnSAT-cores leads to a shift in the way the user faces the proving process. Looking at the (usually few) remaining formulas after `dps-hide` is applied helped the user gain a better understanding on the property to be proved.

This feature was one of the most useful in the verification of the model developed by Zave (2006). The importance of this rule is that it provides a guide in the construction of the proof by revealing those formulas that will be needed to prove the property.

5.2.3 Automated Witness Generation

In this section we will present 4 examples on which we used Dynamite in order to generate witness candidates automatically. We extended Dynamite with a new command solve-inst that, given a sequent whose consequent is a single existentially quantified formula, returns a witness candidate. Besides bounding the total complexity of the generated candidates, it is also possible to bound the number of times each Alloy operator is allowed to occur in generated candidates. In all the experiments each Alloy operator (with the exception of the sequential composition, that was not bounded) was allowed to occur 0 or 1 times. We used a computer with the following configuration: Intel(R) Core(TM) i5 quad core CPU running at 2.67GHz, 8GB of RAM. The operating system was Debian GNU/Linux 6.0, running Kernel 2.6.32-

5-amd64.

Example 1 When verifying assertion `BindingPreservesDeterminism` the following assertion had to be proved:

```
assert prop1 {some ai: Identifier |
      ai in i.^newBinding && a in ai.^(d.dstBinding)}
```

Dynamite retrieves as witness the term

((i.^newBinding) : >(*(d.dstBinding) .a)),

which indeed allowed us to complete the proof. It took Dynamite 172 seconds to retrieve the witness.

Example 2 In (Jackson, 2012, Appendix A), an exercise involving properties of binary relations is proposed. As part of the Alloy model, the following assertion is presented:

```
assert ReformulateNonEmptinessOK {
     all r: univ->univ | some r iff (some x, y: univ | x->y in r) }
```

Let us consider the assertion obtained by:

- skolemizing the universal quantifier,
- substituting `iff` by `implies`, and
- making the antecedent of the implication (`some r`) a new hypothesis.

The resulting model then contains:

```
one sig D {r : univ -> univ}

fact {some D.r}

assert ReformulateNonEmptinessOK { some x, y: univ | x->y in D.r }
```

Notice that there are two quantified variables. Therefore, we applied Dynamite in order to provide first a witness for the outer quantification. Dynamite returned term `(D.(r.univ))` in 161 seconds. Once the witness for the outer quantifier was found, we looked for a witness for the inner quantifier. Since term `(D.(r.univ))` denotes a set, Dynamite produces the following assertion in order to look for the inner witness:

```
fact {x1 in (D.(r.univ))}

assert ReformulateNonEmptinessOK { some y: univ | x1->y in D.r }
```

Dynamite returns term x1.(x1 <: D.r) as the inner witness in 79 seconds. Using these witnesses the assertion is easily verified.

Example 3 The following assertion was presented by Ramananandro (2008) to be analyzed with the aid of the Alloy Analyzer, as part of an Alloy model of the Mondex electronic purse:

```
assert Rbc_init { all c : ConWorld | ConInitState [c] implies
   some b : ConWorld | Rbc [b, c] && BetwInitState [b] }

check Rbc_init for 10 but 2 ConState -- 10007s
check Rbc_init for 10 but 10 ConState -- aborted by user after
                                      -- 7h computation [minisat]
```

According to the original model, it took the Alloy Analyzer 2.8 hours to analyze the assertion using 2 ConState, and the analysis was interrupted after 7 hours for a scope of 10 ConState. We verified this assertion using Dynamite in under 10 minutes. During the proof it was necessary to determine a witness for the existential quantifier in assertion Rbc_init. It took Dynamite 90 seconds to provide the correct witness. The complete proof of the assertion may be seen in Appendix B.

Example 4 This example allows us to show a case in which Dynamite provides a non-atomic coverage as witness candidate. We present the Alloy model including assertion coverSample in Fig. 5.10. Running the witness candidate generator on assertion coverSample returned the coverage {i,i2}. Notice that term i +i2 is not a solution due to fact f2. Let us consider the Alloy instance depicted in Fig. 5.11. Notice first that the instance is indeed a model for the specification. Also, the instance shows that i2 alone cannot be a witness candidate. If we permute i2 and i in Fig. 5.11, we see that i cannot be a witness candidate. It took Dynamite 16 seconds to provide the witness.

5.2.4 Using the Alloy Analyzer as a Proof Helper

Let us see the schematic representation of the proof tree for assertion from Zave's model BindingPreservesReachability (one of the properties we proved), shown in Fig. 5.12. A dps-hyp command was applied in each grey node. It is worth noting that those nodes are the main reason why a branch splitting occurs in that example. This shows that a mistake in the introduction of a case can invalidate a major part of the proof.

A similar situation occurs when a lemma is introduced along a proof as a means to modularize the proof effort. Proof rule dps-lemma calls the Alloy Analyzer in order to analyze whether the introduced lemma is indeed valid.

The experience in using Dynamite on the case study presented here showed us that this feature is a dramatic improvement with respect to the standard case introduction. If a counterexample is found it is shown to the user, so the hypothesis or lemma can be corrected using the information

```
module fm06_extra2
open fm06_defs

sig Agent { }
abstract sig Identifier { }
sig Name, Address extends Identifier { }
sig AddressPair extends Identifier { addr: Address, name: Name }
sig Domain { endpoints: set Agent, space: set Address,
   routing: space −> endpoints }
sig Path { source, dest: Address,
   generator, absorber: Agent }
sig Domain2 extends Domain { dstBinding: Identifier −> Identifier }
{ all i: Identifier | i in dstBinding.Identifier =>
   ( (i in Address => i in space) && (i in AddressPair => i.addr in space)) }
sig Path2 extends Path { origDst: Identifier }

one sig a in Address {}
one sig d in Domain2 {}
sig NB in Identifier { newBinding: set Identifier }
one sig i in Identifier {}
one sig i2 in Identifier {}
one sig i3 in Identifier {}

fact
 { all disj p1, p2: AddressPair | p1.addr != p2.addr || p1.name != p2.name }
fact { some Agent }
fact { some Identifier }
fact { some Domain }
fact { some Path }

fact f1 { some i.^newBinding }
fact f2 { #(i.^newBinding +i2.^newBinding) = 2 }
fact f3 { some i2.^newBinding }
fact f4 { i.^newBinding != i2.^newBinding }

assert coverSample { some x: Identifier | one x.^newBinding }
check coverSample for 6 but 2 Domain
```

Fig. 5.10: A sample model leading to a non-atomic coverage witness.

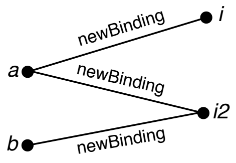


Fig. 5.11: An Alloy instance for assertion coverSample.

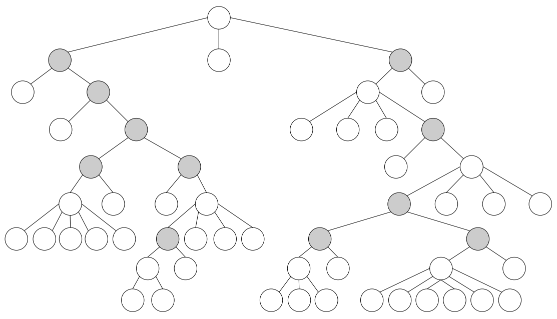


Fig. 5.12: Simplified proof tree for assertion BindingPreservesReachability.

revealed by the counterexample. Counterexamples also give the user a better grasp on the model because they expose tricky corner cases.

6. DISCUSSION, LIMITATIONS & CONCLUSIONS

The analysis of abstractions is a central issue of the software development. Many formal methods had been developed over the years, aimed to attack the problem from different flanks, and using diverse techniques.

Lightweight methods provides the user with automatic techniques to analyze abstractions. Nevertheless, these analysis are usually partial. On the other corner, heavyweight methods provide full analysis but at cost of the need of user-guidance.

In this thesis we tried to bridge the gap between both approaches by presenting a theorem prover for Alloy that complements the analysis provided by the Alloy Analyzer and, at same time, takes advantage of it to make the tool less heavyweight.

A Theorem Prover For Alloy Users. Lightweight analysis methods, such as Alloy, are suitable for a large variety of developments, but critical software needs the highest possible level of confidence. In this thesis we presented a complete calculus that allows one to prove that Alloy assertions follows from a given model. We also embedded this calculus in PVS, achieving an interactive theorem prover for Alloy (Dynamite) in which all the interactions are performed using Alloy syntax. Although it could seem just a aesthetic feature at first glance, it in fact facilitates the usage of the tool by Alloy users.

There are two approaches previous to ours in respect to theorem proving of Alloy assertions. One is the theorem prover Prioni, by Arkoudas et al. (2004). Prioni translates Alloy specifications to first-order formulas characterizing their first-order semantics, and then Athena [Arkoudas (2001)], an interactive theorem prover for first-order logic, is proposed to be used in order to prove the resulting theorem. While the procedure is sound, it is not completely amenable to Alloy users. Switching from a relational to a non-relational language poses an overhead on the user that we are trying to reduce as much as possible.

The other theorem prover is the one presented by Frias et al. (2004). This theorem prover translates Alloy specifications to a close relational language based on binary relations (the calculus for omega closure fork algebras [Frias (2002)]). Since the resulting framework is an equational calculus, quantifiers are removed from Alloy formulas in the translation process. This leads to complicated equations, unnatural for standard Alloy users.

More recent papers [El Ghazi et al. (2011); Ulbrich et al. (2012)] also address the problem of verifying Alloy assertions. El Ghazi et al. (2011) propose the translation of an Alloy model to an SMT-problem, which is solved using the SMT-solver Z3 [de Moura et al. (2008)]. This is a very interesting approach that has some limitations. Complex declarative assertions (as the ones we deal with in this thesis) are unlikely to be solved automatically. Also, the experimental results show that spurious counterexamples can be produced.

An approach close to ours is followed by Ulbrich et al. (2012). They present the Kelloy prover, which is built on top of the KeY first-order theorem prover, by Beckert et al. (2007). Kelloy's embedding into KeY seems to provide greater automation, but no integration with the Alloy Analyzer. In particular, a limitation mentioned in that paper is the need for quantifier instantiation. For some of the examples used in [Ulbrich et al. (2012)], the witness generation technique we introduce here was able to produce correct instantiations, automatically. Another difference arises in the way integers are modeled. While [El Ghazi et al. (2011); Ulbrich et al. (2012)] depart from the Alloy semantics by considering the standard mathematical model of the integers, we stick to the Alloy model where a 2's-complement representation of integers is considered (c.f. Section 3.2.2).

A Lighter Interactive Theorem Prover. We studied a general sequent calculus as an example of a heavyweight analysis procedure and, in Section 2.2.3, outlined some of the points of a proof where insight and creativity are specially required. With these points in mind, we developed novel techniques in which the Alloy Analyzer may be used to help the user develop proofs.

Using SAT-solving in the context of first-order theorem proving is not new. The closest works to Dynamite 1.0 are the thesis by Weber (2008) and the article of Dunets et al. (2010). They follow the idea of our 2007 article [Frias et al. (2007)] of using a model generator to look for counterexamples of formulas being proved by a theorem prover. Previous articles, such as [Weber (2006)], only focus on using the SAT-solver to prove propositional tautologies and use the resolution proofs provided by the SAT-solver to guide the theorem prover proofs. This is more restrictive than Dynamite 1.0 in that Dynamite is not limited to propositional formulas.

Meanwhile, Nitpick [Blanchette et al. (2010)] is used as a counterexample generator for the higher-order logic supported by the Isabelle theorem prover. Nitpick, like Dynamite, is aimed at detecting non-theorems when a proof is initiated. Unlike Dynamite, Nitpick's application seems to be restricted to this case. Similarly, rather than focusing on providing theorem-proving capabilities to a lightweight formal method, Kong et al. (2005) use model checking in order to look for counterexamples before (and during) the theorem proving process. Alternative and more ambitious ways of combining model checking and theorem proving are presented by Shankar (2000). Model checkers and theorem provers interact using the latter for local deductions and propagation of known properties, while the former are employed in order to calculate new properties from reachability predicates or their approximations. Since Alloy models are static, it is not clear how to employ these techniques

Reducing the number of sentences in sequents has been acknowledged as an important problem by the Automated Theorem Proving community. The tool MaLARea [Urban (2007)] reduces sets of hypotheses using machine learning techniques. Sledgehammer [Blanchette et al. (2010)], uses automated theorem provers to select axioms during interactive theorem proving. The iterative technique presented in Section 4.2.4 shows resemblance with [Pudlák (2007)], but [Pudlák (2007)] uses the Darwin model finder tool to convert first-order sentences into function-free clause sets. No notion of UnSAT-cores is provided or used. The SRASS system [Sutcliffe et al. (2007)] uses the ideas presented in [Pudlák (2007)] and complements them with a notion of syntactic relevance, but does not make use of UnSAT-cores. Even members of other communities have faced so similar problems that their solutions can be adapted to our particular interest of trying to find the relevant formulas of a sequent using a model builder. For example, Junker (2004) presents algorithmic procedures to get the most significative constraints from over-constrained problems, in the context of Constraint Programming. These algorithms are general enough to allow one to adapt them in order to find a solution of our hypothesis-selection problem.

The overall experience of proving theorems using Dynamite was very positive. It is remarkable that, although the crucial parts of the proofs are still relying on the user, using the Alloy Analyzer during the proving process proved to be useful in many ways:

- Early detection of errors during key-steps of proofs helped us save time.
- The counterexamples retrieved by the Alloy Analyzer helped us improving our understanding of the problem domain.
- Having leaner sequents helped us focusing on the right proof strategies.
- Using the Alloy language during proofs contributed to smoothing the learning curve.
- Automatically finding witnesses for existentially quantified assertions allowed us to shift the focus to higher-level proof strategies.

Limitations and Threats to Validity. The work reported in this thesis revealed also some limitations of Dynamite in its present state. In the first place, the automation in the proving process is scarce. Only few proof steps are automatically solved (for instance, those referring to typing of relations or to witness candidate generation).

Another limitation this work revealed was the need for an easily portable knowledge base that, as a library of predefined lemmas, allows the use of known general properties in the user-specific proofs. Also, it would be very useful to have a mechanism that proposes which of those lemmas are the most likely to be useful at the current point of the proof. These are areas in which we are currently working.

Despite the case study we worked on, which spotlights were remarked in Chapter 4, it would be necessary to test the usability of Dynamite with a intensive experiment in the field. We believe, as stated in various passages of this thesis, that the characteristic features of Dynamite simplify the proof developing process, but we only tested this assertion with ourselves as the subjects. As with every technique and tool which requires user guidance, this usability experiment is not an easy task. The test must be carefully planed and carried out, in order to get unbiased results.

Concluding remarks. In this thesis we pursue two complementary but distinguishable objectives. In first place, to complement the lightweight approach of the Alloy Analyzer to the problem of analysis of abstractions. We managed to build an interactive theorem prover designed to Alloy users, not only aimed at proving Alloy assertions.

Besides that, we tried to enrich and facilitate the proof process making use of a lightweight approach. We attacked several error prone points of sequent calculus proofs, providing automatic help to the user.

We tested the viability and usefulness of the prover and the helping techniques on recognized case studies. The result of these tests were very promising. The helpful features of Dynamite save uncountable amounts of time, by the early detection of errors and the proposal of existential witnesses.

We believe this thesis serves as a demonstration that by combining different types of techniques, new tools may be developed. Such tools are more accurate, useful and that brings the power of more conclusive analysis to a broader audience.

BIBLIOGRAPHY

Andoni, A., Daniliuc, D., Khurshid, S. and Marinov, D. 2002. *Evaluating the "Small Scope Hypothesis"*, unpublished. Downloadable from `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.72.4000&rep=rep1&type=pdf`.

Arkoudas K. 2001. *Type-ω DPLs.* MIT AI Memo 2001-27.

Arkoudas, K., Khurshid, S., Marinov, D., and Rinard, M. 2004. *Integrating model checking and theorem proving for relational reasoning.* In *Proceedings of the 7th. Conference on Relational Methods in Computer Science (RelMiCS) - 2nd. International Workshop on Applications of Kleene Algebra*, R. Berghammer and B. Möller, Eds. Lecture Notes in Computer Science, vol. 3051. Springer-Verlag, Malente, Germany, 204–213.

Beckert, B., Hahnle, R., and Schmitt, P.H. (eds.). *Verification of Object-Oriented Software: The KeY Approach.* Springer-Verlag (2007).

Bertot, Y. and Castéran, P. 2004. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*, EATCS Texts in Theoretical Computer Science.

Blanchette, J. and Nipkow T. 2010. *Nitpick: A counterexample generator for higher-order logic based on a relational model finder.* Proceedings of the First International Conference on Interactive Theorem Proving (ITP 2010), Lecture Notes in Computer Science 6172, 131–146, Springer.

Clarke, E., Grumberg, O. and Peled, D. 1999. *Model Checking*, MIT Press.

de Moura L. and Bjorner N. 2008. *Z3: An Efficient SMT Solver.* Proceedings of Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2008, Lecture Notes in Computer Science 4963, 337–340, Springer.

Dunets A., Schellhorn G. and Reif W., 2010. *Automated Flaw Detection in Algebraic Specifications*, Journal of Automated Reasoning, 2010.

Eén N. and Sörensson N., 2006. *MiniSat-p-v1.14. A proof-logging version of MiniSat*, `http://minisat.se/MiniSat.html`.

El Ghazi, A. and Taghdiri, M., 2011. *Relational Reasoning via SMT Solving.* Proceedings of Formal Methods (FM) 2011, Lecture Notes in Computer Science 6664, 133-148, Springer.

FRIAS, M. F. 2002. *Fork algebras in algebra, logic and computer science.* Advances in logic, vol. 2. World Scientific Publishing Co., Singapore.

FRIAS M.F., HAEBERER A.M. AND VELOSO P.A.S. 1997. *A Finite Axiomatization for Fork Algebras*, Logic Journal of the IGPL, Vol. 5, No. 3, 311–319.

FRIAS M.F., LÓPEZ POMBO C.G. AND AGUIRRE N. 2004. *A Complete Equational Calculus for Alloy*, in Proceedings of Internacional Conference on Formal Engineering Methods (ICFEM'04), Seattle, USA, November 2004, Lecture Notes in Computer Science 3308, Springer-Verlag, pp. 162–175.

FRIAS, M.F., LÓPEZ POMBO, C.G. AND MOSCATO, M.M. 2007. *Alloy Analyzer+PVS in the analysis and verification of Alloy specifications.* In Grumberg, O., Huth, M., eds.: Proceedings of the 13th. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007). Volume 4424 of Lecture Notes in Computer Science., Braga, Portugal, Springer-Verlag, pp. 587–601.

GENTZEN, G. 1935. Untersuchungen über das logische Schließen. Mathematische Zeitschrift, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North–Holland, 1969.

GORDON, M. J. C. 1989. *Mechanizing Programming Logics in Higher Order Logic.* Current Trends in Hardware Verification and Automated Theorem Proving, Springer-Verlag, 1989.

HOLZMANN, G. J. 2003. *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley.

JACKSON, D. 2012. *Software Abstractions: Logic, Language, and Analysis - Revised version.* The MIT Press.

JACKSON, D. AND DAMON, C. A. 1996. *Elements of style: Analyzying a software design feature with a counterexample detector.* In *EEE Transactions on Software Engineering, 22(7), July 1996.*

JACKSON, D., SHLYAKHTER, I., AND SRIDHARAN, M. 2001. *A micromodularity mechanism.* In *Proceedings of the 8th European software engineering conference held together with the 9th ACM SIGSOFT international symposium on Foundations of software engineering.* Association for the Computer Machinery, ACM Press, Vienna, Austria, 62–73.

JUNKER, U. 2004. *QUICKXPLAIN: Preferred Explanations and Relaxations for Over-constrained Problems.* In *Proceedings of the 19th National*

Conference on Artifical Intelligence AAAI'04. AAAI Press, San Jose, California, USA, 167–172.

Kaufmann, T., McConnell, C., Vazquez, I., Antoniotti, M., Campbell, R., and Amoroso, P. 2002. *ILISP User Manual,* Available at http://sourceforge.net/projects/ilisp/.

Kong, W., Ogata, K., Seino, T. and Futatsugi, K., 2005. *A Lightweight Integration of Theorem Proving and Model Checking for System Verification*, in Proc. of APSEC05, IEEE.

Maddux, R.D. 1991. *Pair-Dense Relation Algebras*, Transactions of the AMS, Vol. 328, N. 1.

Moscato, M., López Pombo, C. G., and Frias, M. F. 2010. *Dynamite 2.0: New Features Based on UnSAT-Core Extraction to Improve Verification of Software Requirements*, International Conference on Theoretical Aspects of Computing (ICTAC) 2010. Lecture Notes in Computer Science 6255, Springer-Verlag, Berlin, Germany, 275–289.

Moscato, M., López Pombo, C. G., and Frias, M. F. 2014. *Dynamite: A Tool for the Verification of Alloy Models Based on PVS*, ACM Transactions On Software Engineering and Methodology (TOSEM), To appear.

Nipkow, T., Paulson, L. C., and Wenzel, M. 2002. *Isabelle/HOL – A proof assistant for higher-order logic.* Lecture Notes in Computer Science, vol. 2283. Springer-Verlag, Berlin, Germany.

Rushby, J. M. 2000. *PVS Bibliography*, Available at: `http://pvs.csl.sri.com/papers/pvs-bib/pvs-bib.dvi`

Owre, S. 2008. *A brief overview of the PVS user interface*, 8th International Workshop User Interfaces for Theorem Provers (UITP08), Montreal, Canada.

Owre, S., Shankar, N., Rushby, J. M., and Stringer-Calvert, D. W. J. 2001. *PVS prover guide*, Version 2.4 ed. Computer Science Laboratory, SRI International.

Owre, S., Rushby, J. M., and Shankar, N., 1992. *PVS: A Prototype Verification System*, Proceedings of the 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence 607, 748–752, Springer.

Pudlák P., 2007. *Semantic Selection of Premisses for Automated Theorem Proving*, in Proceedings of ESARLT 2007, pp. 27–44.

Ramananandro, T., 2008. *Mondex, an electronic purse: specification and refinement checks with the Alloy model-finding method*, Formal Aspects of Computing, 20(1), January 2008, pp. 21–39.

Shankar, N., 2000. *Combining Theorem Proving and Model Checking through Symbolic Analysi*, in Proc. of CONCUR 2000, LNCS, Springer, 2000.

Sutcliffe G. and Puzis Y., 2007. *SRASS a semantic relevance axiom selection system.* `http://www.cs.miami.edu/$\sim$tptp/ATPSystems/SRASS/`.

Torlak E., Chang F. and Jackson D., 2008. *Finding Minimal Unsatisfiable Cores of Declarative Specifications.* Proceedings of Formal Methods 2008, Lecture Notes in Computer Science 5014, 326–341, Springer.

Ulbrich M., Geilmann U., El Ghazi A., and Taghdiri M. 2012. *A Proof Assistant for Alloy Specifications.* Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2012, Lecture Notes in Computer Science 7214, 422–436, Springer.

Weber T., 2008. *MaLARea: a Metasystem for Automated Reasoning in Large Theories*, in Proceedings of ESARLT 2007, pp. 45–58.

Weber T., 2008. *SAT-based Finite Model Generation for Higher-Order Logic*, Ph.D. Thesis, TUM.

Weber T., 2008. *Integrating a SAT Solver with an LCF-style Theorem Prover*, in Proceedings of PDPAR 2005, ENTCS 144(2), pp. 67-78.

Zave, P., 2005. *A Formal Model of Addressing for Interoperating Networks*, in Proceedings of the Thirteenth International Symposium of Formal Methods Europe, Springer-Verlag LNCS 3582, pages 318-333, 2005.

Zave, P. 2006. *Compositional binding in network domains.* In Misra, J., Nipkow, T., Sekerinski, E., eds.: Proceedings of Formal Methods 2006: the 14th. International FME Symposium. Volume 4085 of Lecture Notes in Computer Science., Hamilton, Canada, Springer-Verlag (2006) 332–347

LIST OF SYMBOLS AND CONVENTIONS

Conventions. Unless explicitly stated, the following conventions are used in the whole document: *greek lowercase letters* (α, β, γ, etc.) are used to represent logic formulas, *greek capital letters* (Γ, Δ, etc.) to represent sets of logic formulas, *latin italic lowercase e* (e, e_1, e_2, etc.): expressions, *latin italic capital r s and t* (R, S, T): relations, *latin italic capital a b and c* (A, B, C): sets, *latin italic capital l* (L): languages, *gothic capital letters* ($\mathfrak{A}, \mathfrak{B}$, etc.): PDOCFA structures, *latin italic lowercase v* (v, v_1, v_2, etc.): variables, *latin italic lowercase c* (c, c_1, c_2, etc.): constants, *latin italic lowercase s* (s, s_1, s_2, etc.): Alloy signatures, *latin italic lowercase f* (f, f_1, f_2, etc.): Alloy fields, *latin italic lowercase n* (n, n_1, n_2, etc.): Alloy numeric expressions, *latin italic lowercase d* (d, d_1, d_2, etc.): Alloy declaring expressions.

Symbols. Next, we present a list of symbols used trough the document.

Symbol	*Description*	*Defined in page*
$\mathfrak{i}, \mathfrak{a}, \mathfrak{a}_S$	Alloy instance	18
Names	Set containing all Alloy variable symbols	18
Rel	Collection containing all the relations denotable in Alloy	18
$\mathcal{A}$	Uninterpreted Alloy atoms	18
$\mathcal{Z}$	Numeric Alloy atoms	18
intVal	Bijection between numeric atoms and integers	18
$\mathcal{I}_{\mathsf{S}}$	Legal instances of a specification	19
$\mathcal{I}$	Collection of all Alloy instances	18
$\mathscr{E}^{\mathsf{Alloy}}$	Collection of all Alloy expressions	19
E	Meaning of Alloy expressions	21
$\sim$	Transpose of binary relations	21
R^+	Transitive closure of relational composition (on R)	21
R^*	Reflexive and transitive closure of relational composition	21
$\lhd$	Domain restriction	21
$\rhd$	Range restriction	21
$\oplus$	Relational override	21
$\#$	Cardinality of sets and relations	21
$\models$	Satisfiability of Alloy formulas	24
∇	Fork operator	30
$\star$	Semantics of Fork	30
$\mathscr{E}_L^{\mathsf{PDOCFA}}$	PDOCFA expressions	32
$\mathfrak{p}_{\mathfrak{A}}^L$	Relational instance for a PDOCFA language	31
$\mathscr{I}_{L,\mathfrak{A}}$	Collection of relational instances	31
$\mathcal{X}, \mathcal{X}_{\mathfrak{A}}^L$	Meaning of PDOCFA expressions	32
$\models_{\mathsf{PDOCFA}}$	Satisfiability of PDOCFA formulas	33
$\bullet$	PDOCFA operator for Alloy navigation	38
$\boxtimes$	PDOCFA operator for Alloy Cartesian product	38
$\mathbf{T}, \mathbf{T}_{M \cdot L}$	Translation of Alloy terms	47
$\mathbf{F}, \mathbf{F}_{M \cdot L}$	Translation of Alloy formulas	49

LIST OF FIGURES

APPENDIX

A. COMPLETE PROOFS OF LEMMAS FROM CHAPTER 3

In this appendix we present the proofs of theorems and lemmas used as the basis of the completeness theorem from chapter 3.

From the previous definitions, the following lemma can be proved by induction on the structure of Alloy terms. The lemma, besides being necessary for the proof of interpretability, shows also in what sense the previous constructions can be considered "canonical". Notice that both Alloy and relational instances can be homomorphically extended to functions assigning appropriate values to complex terms built from the constants of each language. Those functions were called E (Def. 3.1.5) and $\mathcal{X}$ (Def. 3.2.3) in chapter 3. For the sake of simplifying the notation we will use in this appendix the same notation for instances and their homomorphic extensions.

Lemma A.0.1: Let $\mathfrak{a}$ be an Alloy instance. Let $\mathfrak{p}$ be a PDOCFA instance defined from $\mathfrak{a}$ according to Def. 3.2.5. Then, for every Alloy term t such that $\mathfrak{a}(t) \subseteq \mathfrak{a}(sig_{i_1}) \times \cdots \times \mathfrak{a}(sig_{i_k})$, we have:

$$\mathfrak{p}_{\mathfrak{F}}(\mathbf{T}(t)) = \begin{cases} \{\, \langle a, a\rangle : a \in \mathfrak{a}(t)\,\}, & \text{if } k = 1 \\ \{\, \langle a_1, a_2 \star \cdots \star a_k\rangle : \langle a_1, a_2, \ldots, a_k\rangle \in \mathfrak{a}(t)\,\}. & \text{if } k > 1 \end{cases}$$

Proof. The proof follows by induction on the structure of the Alloy term t. The proof is trivial for the constants iden, univ and none. We present detailed proofs for the cases in which t is an individual variable or $t = t_1.t_2$. The other cases are easier.

- if t is an individual variable v:
  - if v ranges over atoms:

$$\begin{aligned} \mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(v)) && \text{(by Def. } t\text{)} \\ &= \mathfrak{p}(v) && \text{(by Def. } \mathbf{T}\text{)} \\ &= \{\, \langle \mathfrak{a}(v), \mathfrak{a}(v)\rangle \,\} && \text{(by Def. 3.2.5)} \\ &= \{\, \langle a, a\rangle : a \in \mathfrak{a}(t)\,\} \ . && \text{(by set theory and Def. } t\text{)} \end{aligned}$$

  - if v ranges over higher order relations:

$$\begin{aligned} \mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(v)) && \text{(by Def. } t\text{)} \\ &= \mathfrak{p}(v) && \text{(by Def. } \mathbf{T}\text{)} \\ &= \{\, \langle a_1, a_2 \star \cdots \star a_k\rangle : \langle a_1, a_2, \ldots, a_k\rangle \in \mathfrak{a}(v)\,\} \\ & && \text{(by Def. 3.2.5)} \\ &= \{\, \langle a_1, a_2 \star \cdots \star a_k\rangle : \langle a_1, a_2, \ldots, a_k\rangle \in \mathfrak{a}(t)\,\} \ . \\ & && \text{(by set theory and Def. } t\text{)} \end{aligned}$$

- if $t = t_1.t_2$: Since both the result of the lemma and the definition of $\bullet$ are given by cases, we will consider 6 different cases depending on k_1 (the rank of t_1) and k_2 (the rank of t_2). Following the typing constraints of Alloy, navigation is not defined when $k_1 = k_2 = 1$.

– $k_1 = 1$ and $k_2 = 2$ (then, $k = 1$):

$$\begin{aligned}
\mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1) \bullet \mathbf{T}(t_2)) && \text{(by Def. } \mathbf{T}\text{)}\\
&= \mathfrak{p}(\mathsf{ran}(\mathbf{T}(t_1).\mathbf{T}(t_2))) && \text{(by Def. } \bullet\text{)}\\
&= \mathsf{ran}(\mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2))) && \text{(by } \mathfrak{p} \text{ homomorphism)}\\
&= \mathsf{ran}(\{\, \langle a,a\rangle : a \in \mathfrak{a}(t_1)\,\}\,.\,\{\, \langle a,b\rangle : \langle a,b\rangle \in \mathfrak{a}_S(t_2)\,\})\\
& && \text{(by Ind. Hyp.)}\\
&= \mathsf{ran}(\{\, \langle a,b\rangle : a \in \mathfrak{a}(t_1) \ \wedge\ \langle a,b\rangle \in \mathfrak{a}(t_2)\,\}) && \text{(by Def.“.”)}\\
&= \{\, \langle b,b\rangle : \exists a\,(a \in \mathfrak{a}(t_1) \ \wedge\ \langle a,b\rangle \in \mathfrak{a}(t_2))\,\} && \text{(by Def. } \mathsf{ran}\text{)}\\
&= \{\, \langle b,b\rangle : b \in \mathfrak{a}(t_1)\,.\,\mathfrak{a}(t_2)\,\} && \text{(by Def. “.”)}\\
&= \{\, \langle b,b\rangle : b \in \mathfrak{a}(t_1.\ t_2)\,\} && \text{(by } \mathfrak{a} \text{ homomorphism)}\\
&= \{\, \langle b,b\rangle : b \in \mathfrak{a}(t)\,\}\ . && \text{(by Def. } t\text{)}
\end{aligned}$$

– $k_1 = 1$ and $k_2 > 2$:
Notice that

$$\begin{aligned}
&\mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2))\\
&\ = \{\, \langle a,a\rangle : a \in \mathfrak{a}(t_1)\,\}\\
&\qquad .\,\{\, \langle b_1, b_2 \star \cdots \star b_{k_2}\rangle : \langle b_1, b_2, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t_2)\,\} \quad \text{(by Ind. Hyp.)}\\
&\ = \{\, \langle b_1, b_2 \star \cdots \star b_{k_2}\rangle : b_1 \in \mathfrak{a}(t_1) \wedge \langle b_1, b_2, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t_2)\,\}\ .\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Def.“.”)}
\end{aligned}$$

Then,

$$\begin{aligned}
&\mathsf{ran}(\mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2)))\\
&= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2}\rangle : \exists b_1\,(b_1 \in \mathfrak{a}(t_1) \wedge \langle b_1, b_2, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t_2))\,\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Def. } \mathsf{ran}\text{)}\\
&= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2}\rangle : \langle b_2, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t_1).\mathfrak{a}(t_2)\,\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Def. “.”)}\\
&= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2}\rangle : \langle b_2, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t_1.t_2)\,\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by } \mathfrak{a} \text{ homo.)}\\
&= \{\, \langle b_2 \star \cdots \star b_{k_2}, b_2 \star \cdots \star b_{k_2}\rangle : \langle b_2, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t)\,\}\ . \qquad \text{(by Def. } t\text{)}
\end{aligned}$$

From the definitions of π and ρ, we can reason

$$\begin{aligned}
&\sim \pi\,.\,\mathsf{ran}(\mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2)))\,.\,\rho\\
&\qquad\qquad = \{\, \langle b_2, b_3 \star \cdots \star b_{k_2}\rangle : \langle b_2, b_3, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t)\,\}\ . \qquad \text{(A.1)}
\end{aligned}$$

Joining the previous proofs we obtain:

$$\begin{aligned}
\mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(t_1.t_2)) && \text{(by Def. } \mathbf{T}\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1) \bullet \mathbf{T}(t_2)) && \text{(by Def. } \mathbf{T}\text{)}\\
&= \mathfrak{p}(\sim \pi\,.\,\mathsf{ran}(\mathbf{T}(t_1).\mathbf{T}(t_2))\,.\,\rho) && \text{(by Def. } \bullet\text{)}\\
&=\sim \pi\,.\,\mathsf{ran}(\mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2)))\,.\,\rho && \text{(by } \mathfrak{p} \text{ homomorphism)}\\
&= \{\, \langle b_2, b_3 \star \cdots \star b_{k_2}\rangle : \langle b_2, b_3, \ldots, b_{k_2}\rangle \in \mathfrak{a}(t)\,\}\ . && \text{(by (A.1))}
\end{aligned}$$

– $k_1 = 2$ and $k_2 = 1$ (then, $k = 1$):

$$\begin{aligned}
\mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1) \bullet \mathbf{T}(t_2)) && \text{(by Def. } \mathbf{T}\text{)}\\
&= \mathfrak{p}(\mathsf{dom}(\mathbf{T}(t_1).\mathbf{T}(t_2))) && \text{(by Def. } \bullet\text{)}\\
&= \mathsf{dom}(\mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2))) && \text{(by } \mathfrak{p} \text{ homomorphism)}\\
&= \mathsf{dom}(\{\, \langle a,b\rangle : \langle a,b\rangle \in \mathfrak{a}(t_1) \,\} \,.\, \{\, \langle b,b\rangle : b \in \mathfrak{a}(t_2) \,\})\\
& && \text{(by Ind. Hyp.)}\\
&= \mathsf{dom}(\{\, \langle a,b\rangle : \langle a,b\rangle \in \mathfrak{a}(t_1) \ \wedge\ b \in \mathfrak{a}(t_2) \,\}) && \text{(by Def.``.'')}\\
&= \{\, \langle a,a\rangle : \exists b\, (\langle a,b\rangle \in \mathfrak{a}(t_1) \ \wedge\ b \in \mathfrak{a}(t_2)) \,\} && \text{(by Def. } Dom\text{)}\\
&= \{\, \langle a,a\rangle : a \in \mathfrak{a}(t_1).\mathfrak{a}(t_2) \,\} && \text{(by Def. ``.'')}\\
&= \{\, \langle a,a\rangle : a \in \mathfrak{a}(t_1.t_2) \,\} && \text{(by } e \text{ homomorphism)}\\
&= \{\, \langle a,a\rangle : a \in \mathfrak{a}(t) \,\} \ . && \text{(by Def. } t\text{)}
\end{aligned}$$

– $k_1 = 2$ and $k_2 > 1$:

$$\begin{aligned}
\mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1) \bullet \mathbf{T}(t_2)) && \text{(by Def. } \mathbf{T}\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1).\mathbf{T}(t_2)) && \text{(by Def. } \bullet\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1)).\mathfrak{p}(\mathbf{T}(t_2)) && \text{(by } \mathfrak{p} \text{ homomorphism)}\\
&= \{\, \langle a,a_1\rangle : \langle a,a_1\rangle \in \mathfrak{a}(t_1) \,\}\\
&\qquad .\, \{\, \langle a_1, a_2 \star \cdots \star a_{k_2}\rangle : \langle a_1,a_2,\ldots,a_{k_2}\rangle \in \mathfrak{a}(t_2) \,\}\\
& && \text{(by Ind. Hyp.)}\\
&= \{\langle a, a_2 \star \cdots \star a_{k_2}\rangle : \exists a_1\, (\langle a,a_1\rangle \in \mathfrak{a}(t_1)\\
&\qquad \wedge\ \langle a_1,a_2,\ldots,a_{k_2}\rangle \in \mathfrak{a}(t_2))\} && \text{(by Def. ``.'')}\\
&= \{\, \langle a, a_2 \star \cdots \star a_{k_2}\rangle : \langle a,a_2,\ldots,a_{k_2}\rangle \in \mathfrak{a}(t_1).\mathfrak{a}(t_2) \,\}\\
& && \text{(by Def. ``.'')}\\
&= \{\, \langle a, a_2 \star \cdots \star a_{k_2}\rangle : \langle a,a_2,\ldots,a_{k_2}\rangle \in e(t_1.t_2) \,\}\\
& && \text{(by } \mathfrak{a} \text{ homo.)}\\
&= \{\, \langle a, a_2 \star \cdots \star a_{k_2}\rangle : \langle a,a_2,\ldots,a_{k_2}\rangle \in \mathfrak{a}(t) \,\} \ . && \text{(by Def. } t\text{)}
\end{aligned}$$

– $k_1 > 2$ and $k_2 = 1$:

$$\begin{aligned}
\mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(\mathbf{T}(t_1.t_2)) && \text{(by Def. } t\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1) \bullet \mathbf{T}(t_2)) && \text{(by Def. } \mathbf{T}\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1).\,(\mathsf{iden}\otimes(\cdots\otimes((\mathsf{iden}\otimes\mathbf{T}(t_2))\,.\pi)))) && \text{(by Def. } \bullet\text{)}\\
&= \mathfrak{p}(\mathbf{T}(t_1)).\,(\mathsf{iden}\otimes(\cdots\otimes((\mathsf{iden}\otimes\mathfrak{p}(\mathbf{T}(t_2)))\,.\pi))) \ .\\
& && \text{(by } \mathfrak{p} \text{ homo.)}
\end{aligned}$$

Notice that

$$\begin{aligned}
\mathsf{iden}\otimes\mathfrak{p}(\mathbf{T}(t_2)) &= \{\, \langle a\star b, a\star b\rangle : \langle b,b\rangle \in \mathfrak{p}(\mathbf{T}(t_2)) \,\} && \text{(by Def. } \otimes\text{)}\\
&= \{\, \langle a\star b, a\star b\rangle : b \in \mathfrak{a}(t_2) \,\} \ . && \text{(by Ind. Hyp.)}
\end{aligned}$$

Then,

$$(\mathsf{iden}\otimes\mathfrak{p}(\mathbf{T}(t_2)))\,.\pi = \{\, \langle a\star b, a\rangle : b \in \mathfrak{a}(t_2) \,\} \ .$$

Therefore,

$$\begin{aligned}&\mathsf{iden}\otimes(\cdots\otimes(\mathsf{iden}\otimes\mathfrak{p}(\mathbf{T}(t_2)))\,.\pi) = \\ &\quad\{\,\langle a_1\star\cdots\star a_{k_1-3}\star a\star b, a_1\star\cdots\star a_{k_1-2}\star a\rangle : b\in\mathfrak{a}(t_2)\,\}\ .\end{aligned}\tag{A.2}$$

By inductive hypothesis,

$$\mathfrak{p}(\mathbf{T}(t_1)) = \{\,\langle b_1, b_2\star\cdots\star b_{k_1}\rangle : \langle b_1, b_2,\ldots,b_{k_1}\rangle\in\mathfrak{a}(t_1)\,\}\ .\tag{A.3}$$

From (A.2) and (A.3), $\langle c_1, c_2\star\cdots\star c_{k_1-1}\rangle\in\mathfrak{p}(\mathbf{T}(t))$ iff there exists (due to the definition of composition of binary relations) an object $d_1\star\cdots\star d_{k_1-1}$ such that:

$*$ $\langle c_1, d_1\star\cdots\star d_{k_1-1}\rangle\in\mathfrak{p}(\mathbf{T}(t_1))$, (or, equivalently, because of the relationship between $\mathfrak{p}$ and $\mathfrak{a}$: $\langle c_1, d_1,\ldots,d_{k_1-1}\rangle\in\mathfrak{a}(t_1)$),

$*$ $d_{k_1-1}\in\mathfrak{a}(t_2)$, and

$*$ $d_i = c_{i+1}$ for $1\le i\le k_1-2$.

From the previous conditions,

$$\begin{aligned}&\langle c_1, c_2\star\cdots\star c_{k_1-1}\rangle\in\mathfrak{p}(\mathbf{T}(t))\\ \textit{iff}\ &\exists d_{k_1-1}\,(\langle c_1, c_2,\ldots,c_{k_1-1}, d_{k_1-1}\rangle\in\mathfrak{a}(t_1)\text{ and }d_{k_1-1}\in\mathfrak{a}(t_2))\\ \textit{iff}\ &\langle c_1, c_2,\ldots,c_{k_1-1}\rangle\in\mathfrak{a}(t_1)\,.\,\mathfrak{a}(t_2)\\ \textit{iff}\ &\langle c_1, c_2,\ldots,c_{k_1-1}\rangle\in e(t_1\,.\,t_2)\\ \textit{iff}\ &\langle c_1, c_2,\ldots,c_{k_1-1}\rangle\in\mathfrak{a}(t)\ .\end{aligned}$$

Thus, $\mathfrak{p}(\mathbf{T}(t)) = \{\,\langle c_1, c_2\star\cdots\star c_{k_1-1}\rangle : \langle c_1, c_2,\ldots,c_{k_1-1}\rangle\in\mathfrak{a}(t)\,\}$.

– $k_1 > 2$ and $k_2 > 1$:

$$\begin{aligned}\mathfrak{p}(\mathbf{T}(t)) &= \mathfrak{p}(T(t_1\,.\,t_2)) && \text{(by Def. } t\text{)}\\ &= \mathfrak{p}(\mathbf{T}(t_1)\bullet\mathbf{T}(t_2)) && \text{(by Def. }\mathbf{T}\text{)}\\ &= \mathfrak{p}(\mathbf{T}(t_1))\bullet\mathfrak{p}(\mathbf{T}(t_2))\ . && \text{(by }\mathfrak{p}\text{ homo.)}\\ &= \mathfrak{p}(\mathbf{T}(t_1).\,(\mathsf{iden}\otimes(\cdots\otimes(\mathsf{iden}\otimes\mathbf{T}(t_2))))) && \text{(by Def. }\bullet\text{)}\\ &= \mathfrak{p}(\mathbf{T}(t_1)).\,(\mathsf{iden}\otimes(\cdots\otimes(\mathsf{iden}\otimes\mathfrak{p}(\mathbf{T}(t_2))))) && \text{(by }\mathfrak{p}\text{ homo.)}\end{aligned}$$

By inductive hypothesis,

$$\mathfrak{p}(\mathbf{T}(t_1)) = \{\,\langle a_1, a_2\star\cdots\star a_{k_1}\rangle : \langle a_1, a_2,\ldots,a_{k_1}\rangle\in\mathfrak{a}(t_1)\,\}\tag{A.4}$$

and

$$\mathfrak{p}(\mathbf{T}(t_2)) = \{\,\langle b_1, b_2\star\cdots\star b_{k_2}\rangle : \langle b_1, b_2,\ldots,b_{k_2}\rangle\in\mathfrak{a}(t_2)\,\}\ .\tag{A.5}$$

From (A.5) and Def. $\otimes$,

$$\begin{aligned}&\mathsf{iden}\otimes(\cdots\otimes(\mathsf{iden}\otimes\mathfrak{p}(\mathbf{T}(t_2))))\\ &\quad=\{\langle a_1\star\cdots\star a_{k_1-2}\star b_1, a_1\star\cdots\star a_{k_1-2}\star b_2\star\cdots\star b_{k_2}\rangle :\\ &\qquad\qquad\langle b_1, b_2,\ldots,b_{k_2}\rangle\in\mathfrak{a}(t_2)\}\ .\end{aligned}\tag{A.6}$$

From (A.4) and (A.6), $\langle c_1, c_2\star\cdots\star c_{k_1+k_2-2}\rangle\in\mathfrak{p}(\mathbf{T}(t))$ iff there exists (due to the definition of composition of binary relations) $d_1,\ldots,d_{k_1-1}$ such that:

  - $\langle c_1, d_1 \star \cdots \star d_{k_1-1} \rangle \in \mathfrak{p}(\mathbf{T}(t_1))$ (or, equivalently as aforementioned, $\langle c_1, d_1, \ldots, d_{k_1-1} \rangle \in \mathfrak{a}(t_1)$),
  - $\langle d_{k_1-1}, c_{k_1}, \ldots, c_{k_1+k_2-2} \rangle \in \mathfrak{a}(t_2)$, and
  - $d_i = c_{i+1}$ for $1 \leq i \leq k_1 - 2$.

  From the previous conditions,

$$\begin{aligned}
&\langle c_1, c_2 \star \cdots \star c_{k_1+k_2-2} \rangle \in e'(\mathbf{T}(t)) \\
\textit{iff}\ &\exists d_{k_1-1}\,(\langle c_1, c_2, \ldots, c_{k_1-1}, d_{k_1-1} \rangle \in \mathfrak{a}(t_1) \\
&\quad \text{and } \langle d_{k_1-1}, c_{k_1}, \ldots, c_{k_1+k_2-2} \rangle \in \mathfrak{a}(t_2)) \\
\textit{iff}\ &\langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in \mathfrak{a}(t_1)\,.\,\mathfrak{a}(t_2) \\
\textit{iff}\ &\langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in e(t_1\,.\,t_2) \\
\textit{iff}\ &\langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in \mathfrak{a}(t)\ .
\end{aligned}$$

  Thus,

$$\mathfrak{p}(\mathbf{T}(t)) = \{\, \langle c_1, c_2 \star \cdots \star c_{k_1+k_2-2} \rangle : \langle c_1, c_2, \ldots, c_{k_1+k_2-2} \rangle \in \mathfrak{a}(t) \,\}$$

■

Lemma A.0.2: Let L be a PDOCFA language, $\mathfrak{F}$ a proper PDOCFA and $\mathfrak{p}^L_{\mathfrak{F}}$ a relational instance assigning values from $\mathfrak{F}$ to the constant and variable symbols of L. The Alloy instance $\mathfrak{a}$, built according to Def. 3.2.6, satisfies for every Alloy term t with $\mathfrak{a}(t) \subseteq \mathfrak{a}(sig_{i_1}) \times \cdots \times \mathfrak{a}(sig_{i_k})$:

- if $k = 1$, $\mathfrak{a}(t) = \{\, a : \langle a, a \rangle \in \mathfrak{p}(\mathbf{T}(t)) \,\}$,
- if $k > 1$, $\mathfrak{a}(t) = \{\, \langle a_1, a_2, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in \mathfrak{p}(\mathbf{T}(t)) \,\}$.

Proof. By Def. 3.2.6, $\mathfrak{a}$ satisfies:

- for every partial identity symbol $\mathsf{iden}_s \in L$,
$$\mathfrak{a}(s) = \{\, a : \langle a, a \rangle \in \mathfrak{p}(\mathsf{iden}_s) \,\}\ ,$$
- for each constant symbol c, nor representing a comprehension expression nor being general constants (such as: univ, iden, none, iden_U, $\mathsf{iden}_\mathbb{Z}$),
$$\mathfrak{a}(c) = \{\, \langle a_1, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in \mathfrak{p}(c) \,\}$$
  where $k = rank(\mathcal{X}(c)\mathfrak{p})$,
- for every variable symbol v, if v is representing an Alloy variable ranging over atoms,
$$\mathfrak{a}(v_i) = a \text{ such that } \mathfrak{p}(v_i) = \{\, \langle a, a \rangle \,\}$$
  else, calling k the rank codified in v,
$$\mathfrak{a}(v) = \{\, \langle a_1, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in \mathfrak{p}(v) \,\}\ .$$

From the definition of $\mathfrak{a}$, $\mathfrak{p}$ satisfies:

- $\mathfrak{p}(\mathsf{iden}_s) = \{\, \langle a, a \rangle : a \in \mathfrak{a}(s) \,\}$,

- $\mathfrak{p}(c) = \{\, \langle a_1, a_2 \star \cdots \star a_n \rangle : \langle a_1, \ldots, a_n \rangle \in \mathfrak{a}(c) \,\}$,
- $\mathfrak{p}(v) = \begin{cases} \{\, \langle a, a \rangle \,\} \text{ s.t. } \mathfrak{a}(v) = a & \text{if } v \text{ is representing an Alloy variable ranging over atoms} \\ \{\, \langle a_1, a_2 \star \cdots \star a_k \rangle : \langle a_1, a_2, \ldots, a_k \rangle \in \mathfrak{a}(v) \,\} & \text{otherwise.} \end{cases}$

We now have an Alloy instance $\mathfrak{a}$ and a relational instance $\mathfrak{p}$ that satisfy the conditions in Def. 3.2.5. Notice then that $\mathfrak{F}$ is compatible with the Alloy instance $\mathfrak{a}$. From Lemma A.0.1, for every Alloy term t with $\mathfrak{a}(t) \subseteq \mathfrak{a}(sig_{i_1}) \times \cdots \times \mathfrak{a}(sig_{i_k})$:

- if $k = 1$, $\mathfrak{p}(\mathbf{T}(t)) = \{\, \langle a, a \rangle : a \in \mathfrak{a}(t) \,\}$,
- if $k > 1$, $\mathfrak{p}(\mathbf{T}(t)) = \{\, \langle a_1, a_2 \star \cdots \star a_k \rangle : \langle a_1, a_2, \ldots, a_k \rangle \in \mathfrak{a}(t) \,\}$.

It then follows that, for every such term:

- if $k = 1$, $\mathfrak{a}(t) = \{\, a : \langle a, a \rangle \in \mathfrak{p}(\mathbf{T}(t)) \,\}$,
- if $k > 1$, $\mathfrak{a}(t) = \{\, \langle a_1, a_2, \ldots, a_k \rangle : \langle a_1, a_2 \star \cdots \star a_k \rangle \in \mathfrak{p}(\mathbf{T}(t)) \,\}$.

■

The following lemma will be used in the proof of Lemma A.0.4.

Lemma A.0.3: Let $\mathfrak{a}$ be an Alloy instance. Let $\mathfrak{p}$ be a relational instance defined from $\mathfrak{a}$, according to Def. 3.2.5. Let $\mathfrak{F}_{\mathfrak{a}}$ be a PDOCFA compatible with instance $\mathfrak{a}$ (c.f. Def. 3.2.7).

Let a be an atom from a signature s, $\mathfrak{a}' = (\mathfrak{a} \oplus [x \mapsto a])$ an Alloy instance that can only differ from $\mathfrak{a}$ in the value of the variable x, $\mathfrak{p}'$ a relational instance defined from $\mathfrak{a}$', and $\mathfrak{F}_{\mathfrak{a}'}$ a PDOCFA compatible with $\mathfrak{a}'$.

Finally, let $r = \{\, \langle a, a \rangle \,\} \subseteq iden_s$ be a point. Then,

$$\mathfrak{F}_{\mathfrak{a}}, \mathfrak{p} \oplus \{x \mapsto r\} \models_{\mathsf{PDOCFA}} \beta \quad \textit{iff} \quad \mathfrak{F}_{\mathfrak{a}'}, \mathfrak{p}' \models_{\mathsf{PDOCFA}} \beta$$

Proof. In order to prove the lemma it suffices to show that $\mathfrak{F}_{\mathfrak{a}} = \mathfrak{F}_{\mathfrak{a}'}$ and $\mathfrak{p} \oplus \{x \mapsto r\} = \mathfrak{p}'$. According to Defs. 3.2.5 and 3.2.7, the construction of algebra $\mathfrak{F}_{\mathfrak{a}}$ does not depend on the value $\mathfrak{a}$ assigns to variables. Therefore, it is immediate that $\mathfrak{F}_{\mathfrak{a}} = \mathfrak{F}_{\mathfrak{a}'}$.

On the other hand, for all signatures, fields and variables distinct of x, it is clear that $\mathfrak{p} \oplus \{x \mapsto r\}$ and $\mathfrak{p}'$ agree. For variable x we have:

$$(\mathfrak{p} \oplus \{x \mapsto r\})(x) = r = \{\, \langle a, a \rangle \,\} \ .$$

Similarly, by Def. 3.2.5,

$$\begin{aligned} \mathfrak{p}'(x) &= \{\, \langle \mathfrak{a}'(x), \mathfrak{a}'(x) \rangle \,\} \\ &= \{\, \langle (\mathfrak{a} \oplus [x \mapsto a])\,(x), (\mathfrak{a} \oplus [x \mapsto a])\,(x) \rangle \,\} \\ &= \{\, \langle a, a \rangle \,\} \ . \end{aligned}$$

Then, $\mathfrak{p} \oplus \{x \mapsto r\}$ and $\mathfrak{p}'$ also agree for x and hence $\mathfrak{p} \oplus \{x \mapsto r\} = \mathfrak{p}'$. ■

Lemma A.0.4: Let $\mathfrak{a}$ be an Alloy instance. Let $\mathfrak{p}$ be defined from $\mathfrak{a}$ according to Def. 3.2.5. Let $\mathfrak{F}$ be a PDOCFA compatible with instance $\mathfrak{a}$ (c.f. Def. 3.2.7). Then, being φ an Alloy formula interpretable in $\mathfrak{a}$,

$$\mathfrak{a} \models \varphi \quad \textit{iff} \quad \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \mathbf{F}(\varphi)$$

Proof. The proof proceeds by induction on the structure of the Alloy formula φ. We will concentrate on formulas built from atomic formulas (inclusions) and existential quantification. The other cases are easier.

- $\varphi = t_1 \text{ in } t_2$:

  For φ to be well-formed, t_1 and t_2 must stand for relations of the same arity.

$$\begin{array}{lll} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \mathbf{F}(t_1 \text{ in } t_2) & \\ \textit{iff} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \mathbf{T}(t_1) \text{ in } \mathbf{T}(t_2) & \text{(by Def. 3.2.12)} \\ \textit{iff} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \mathbf{T}(t_2) = \mathbf{T}(t_1) \ + \ \mathbf{T}(t_2) & \text{(by Notation remark 5)} \\ \textit{iff} & \mathfrak{p}(\mathbf{T}(t_2)) = \mathfrak{p}(\mathbf{T}(t_1)) \ + \ \mathfrak{p}(\mathbf{T}(t_2)) & \text{(by Def. of } \models_{\mathsf{PDOCFA}} \text{ and } \mathfrak{p} \text{ homo.)} \\ \textit{iff} & \mathfrak{p}(\mathbf{T}(t_1)) \subseteq \mathfrak{p}(\mathbf{T}(t_2)) \ . & \text{(by set-theory)} \end{array}$$

  There are now two possibilities, namely, either both t_1, t_2 have arity 1, or they both have arity $k > 2$. Let us continue the proof for the unary case, being the remaining case similar. We then have

$$\begin{array}{lll} & \mathfrak{p}(\mathbf{T}(t_1)) \subseteq \mathfrak{p}(\mathbf{T}(t_2)) & \\ \textit{iff} & \{ \langle a,a \rangle : a \in \mathfrak{a}(t_1) \} \subseteq \{ \langle a,a \rangle : a \in \mathfrak{a}(t_2) \} & \text{(by Lemma A.0.1)} \\ \textit{iff} & \mathfrak{a}(t_1) \subseteq \mathfrak{a}(t_2) & \text{(by set-theory)} \\ \textit{iff} & \mathfrak{a} \models t_1 \text{ in } t_2 \ . & \text{(by Def. of } \models) \end{array}$$

- Lets face now the case in which φ is an existential quantification. In particular we will show the case of a quantification ranging over atoms. The higher order cases are very similar to this one. Let suppose that

$$\varphi = \text{some } x : \text{one } s \ \mid \ \alpha$$

  Then,

$$\begin{array}{lll} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \mathbf{F}(\text{some } x : \text{one } s \ \mid \ \alpha) & \\ \textit{iff} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \text{some } \ x \mid \ \mathbf{F}(x \text{ in} s \ \&\& \ \mathcal{R}(x, \text{one } s) \ \&\& \ \alpha) & \text{(by Def. of } \models_{\mathsf{PDOCFA}}) \\ \textit{iff} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \text{some } \ x \mid \ \mathbf{F}(x \text{ in} s \ \&\& \ \text{one } x \ \&\& \ \alpha) & \text{(by } \mathcal{R}\text{, from Def. 3.1.6)} \\ \textit{iff} & \mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \text{some } \ x \mid \ x \text{ in iden}_s \ \&\& \ \text{one}(x) \ \&\& \ \mathbf{F}(\alpha) & \\ & & \text{(by } \mathbf{T} \text{ and } \mathbf{F}\text{, from Def. 3.2.11 and 3.2.12)} \\ \textit{iff} & \text{there exists } R \in \mathbf{R}_{\mathfrak{F}} \text{ such that:} & \\ & \quad \mathfrak{F}, \mathfrak{p} \oplus \{x \mapsto R\} \models_{\mathsf{PDOCFA}} x \text{ in iden}_s \ \text{ and} & \\ & \quad \mathfrak{F}, \mathfrak{p} \oplus \{x \mapsto R\} \models_{\mathsf{PDOCFA}} \text{one}(x) \ \text{ and} & \\ & \quad \mathfrak{F}, \mathfrak{p} \oplus \{x \mapsto R\} \models_{\mathsf{PDOCFA}} \mathbf{F}(\alpha) \ . & \text{(by Def. of } \models_{\mathsf{PDOCFA}}) \end{array}$$

  It is easy to prove that such R is, in fact, a point included in $iden_s$. Let us assume $R = \{ \langle a,a \rangle \}$ (for some $a \in \mathfrak{a}(s)$). Notice that algebra $\mathfrak{F}$ does not depend on the value $\mathfrak{a}$ assigns to variables. Thus, $\mathfrak{F}$ is also compatible with instance $\mathfrak{a} \oplus \{x \mapsto a\}$. Moreover, from the proof of Lemma A.0.3, if we call $\mathfrak{p}'$ an instance defined from $\mathfrak{a} \oplus \{x \mapsto a\}$ (according to Def. 3.2.5), $\mathfrak{p}' = \mathfrak{p} \oplus \{x \mapsto R\}$.

Thus,

$$\begin{aligned}
&\text{there exists } R \in \mathbf{R}_{\mathfrak{F}} \text{ such that:} \\
&\quad \mathfrak{F}, \mathfrak{p} \oplus \{x \mapsto R\} \models_{\mathsf{PDOCFA}} x \ \mathsf{in}\ \mathsf{iden}_s \ \text{ and} \\
&\quad \mathfrak{F}, \mathfrak{p} \oplus \{x \mapsto R\} \models_{\mathsf{PDOCFA}} \mathsf{one}(x) \ \text{ and} \\
&\quad \mathfrak{F}, \mathfrak{p} \oplus \{x \mapsto R\} \models_{\mathsf{PDOCFA}} \mathbf{F}(\alpha) \ . && \text{(by Def. of } \models_{\mathsf{PDOCFA}}) \\
\textit{iff}\ &\text{there exists } a \in \mathfrak{a}(s) \text{ such that} \\
&\quad \mathfrak{F}, \mathfrak{p}' \models_{\mathsf{PDOCFA}} \mathbf{F}(x \ \mathsf{in} s \ \&\& \ \mathsf{one}\ x \ \&\& \ \alpha) \\
& && \text{(with } \mathfrak{p}' \text{ as in the previous comment)} \\
\textit{iff}\ &\text{there exists } a \in \mathfrak{a}(s) \text{ such that} \\
&\quad \mathfrak{a} \oplus \{x \mapsto a\} \models x \ \mathsf{in} s \ \&\& \ \mathsf{one}\ x \ \&\& \ \alpha && \text{(by inductive hypothesis)} \\
\textit{iff}\ &\mathfrak{a} \models \mathsf{some}\ x\text{: } \mathsf{one}\ s \ \mid \ \alpha && \text{(by Def. of } \models)
\end{aligned}$$

The rest of the cases to be proven can be solved following similar strategies. ■

Lemma A.0.5: Let $\mathfrak{F}$ be a proper PDOCFA. Let $\mathfrak{p}$ be a relational instance. Then, there exists an Alloy instance $\mathfrak{a}$ built according to Def. 3.2.6 such that for every Alloy formula φ,

$$\mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} \varphi \quad \textit{iff} \quad \mathfrak{a} \models \varphi \ .$$

Proof. The proof proceeds by induction on the structure of formula φ.

- $\varphi = t_1$ in t_2:

$$\begin{aligned}
\mathfrak{F}, \mathfrak{p} \models_{\mathsf{PDOCFA}} t_1 \ \mathsf{in}\ t_2 \ \textit{iff}\ \mathfrak{F}, \mathfrak{p} &\models_{\mathsf{PDOCFA}} \mathbf{T}(t_1) \ \mathsf{in}\ \mathbf{T}(t_2) && \text{(by Def. 3.2.12)} \\
&\iff \mathfrak{p}(\mathbf{T}(t_1)) \subseteq \mathfrak{p}(\mathbf{T}(t_2)) && \text{(as in previous proof)} \\
&\iff \mathfrak{a}(t_1) \subseteq \mathfrak{a}(t_2) && \text{(by Lemma A.0.2)} \\
&\iff \mathfrak{a} \models t_1 \ \mathsf{in}\ t_2 \ . && \text{(by Def. of } \models)
\end{aligned}$$

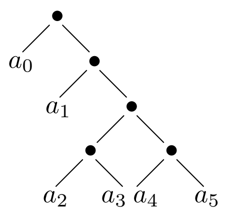
The remaining parts of the proof follow a structure similar to that of the proof of Lemma A.0.4. ■

Lemma A.0.6: Given an Alloy instance $\mathfrak{a}$, there exists a proper PDOCFA $\mathfrak{F}$ compatible with $\mathfrak{a}$.

Proof. Assume the Alloy model declares signatures $Sig_1, \ldots, Sig_I$. Let $A = \bigcup_{1 \leq j \leq I} e(Sig_j)$. Let $Tree(A)$ be the smallest set satisfying the following conditions:

- $A \subseteq Tree(A)$, and
- $Tree(A) \times Tree(A) \subseteq Tree(A)$.

$Tree(A)$ describes finite binary trees with data from A in the leaves. For instance, element $\langle a_0, \langle a_1, \langle\langle a_2, a_3\rangle, \langle a_4, a_5\rangle\rangle\rangle\rangle \in Tree(A)$ describes the tree



Let us consider the PDOCFA $\mathfrak{F}$ with universe[1] $Pw(\mathit{Tree}(A) \times \mathit{Tree}(A))$. All the operators but fork have their standard set-theoretical meaning. For fork we define

$$R \nabla S = \{ \langle a, \langle b, c \rangle \rangle : \langle a, b \rangle \in R \;\&\&\; \langle a, c \rangle \in S \} \ .$$

In order to prove compatibility we must show that signatures and fields defined in the Alloy model are given appropriate values according to instance $\mathfrak{p}$, the relational instance defined from $\mathfrak{a}$. For signature Sig_j $(1 \leq j \leq I)$, $\mathfrak{p}(\mathsf{iden}_{Sig_j}) = iden_{Sig_j}$, which clearly belongs to $Pw(\mathit{Tree}(A) \times \mathit{Tree}(A))$. For a field F declared as

```
sig S { F : S1->...->Sk }
```

the relation $\mathfrak{p}(F)$ defined as

$$\begin{aligned}&\{\langle s_0, s_1 \star \cdots \star s_k \rangle : \\ &\qquad s_0 \in S \;\&\&\; s_1 \in S_1 \;\&\&\; \cdots \&\& \; s_k \in S_k \;\&\&\; \langle s_0, s_1, \ldots, s_k \rangle \in \mathfrak{a}(F)\}\end{aligned}$$

belongs to $Pw(\mathit{Tree}(A) \times \mathit{Tree}(A))$ provided $a \star b$ is defined as $\langle a, b \rangle$.

■

[1] We denote by $Pw(X)$ the power set of set X.

B. CASE STUDY: PROOF OF MONDEX PROPERTY

In this section we provide the complete proof of a property from the Alloy model for Mondex electronic purse by Ramananandro (2008), as an example of use of Dynamite. The property is stated as an Alloy assertion called `Rbc_init`.

```
Rbc_init :

  |-------
{1}   (some b : ConWorld| ((Rbc[b, c]) && (BetwInitState[b])))

Rule? (rewrite-msg-off)
Turning off rewriting commentary,
No change on: (rewrite-msg-off)
Rbc_init :

  |-------
{1}   (some b : ConWorld| ((Rbc[b, c]) && (BetwInitState[b])))

Rule? (use "Rbc_pre1")
Using lemma Rbc_pre1,
this simplifies to:
Rbc_init :

{-1}  ((((((Concrete[c]) && (BetwInitState[cb]))
          && ((c . conAuthPurse) = (cb . conAuthPurse)))
         && (XiConPurse[c, cb, (c . conAuthPurse)]))
        && ((c . archive) = (cb . archive)))
       && ((c . ether) in (cb . ether)))
  |-------
[1]   (some b : ConWorld| ((Rbc[b, c]) && (BetwInitState[b])))

Rule? (prop)
Applying propositional simplification,
this simplifies to:
Rbc_init :

{-1}  (Concrete[c])
{-2}  (BetwInitState[cb])
{-3}  ((c . conAuthPurse) = (cb . conAuthPurse))
{-4}  (XiConPurse[c, cb, (c . conAuthPurse)])
{-5}  ((c . archive) = (cb . archive))
{-6}  ((c . ether) in (cb . ether))
  |-------
[1]   (some b : ConWorld| ((Rbc[b, c]) && (BetwInitState[b])))
```

```
-- AT THIS POINT WE USE THE WITNESS CANDIDATE POSTULATION ROUTINE
-- WHICH SUGGESTS cb AS THE WITNESS.
Rule? (inst 1 "cb")
Instantiating the top quantifier in 1 with the terms: cb,
this yields  2 subgoals:
Rbc_init.1 :

[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   ((Rbc[cb, c]) && (BetwInitState[cb]))

Rule? (prop)
Applying propositional simplification,
this simplifies to:
Rbc_init.1 :

[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   (Rbc[cb, c])

Rule? (expand "Rbc")
Expanding the definition of Rbc,
this simplifies to:
Rbc_init.1 :

[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   (((((cb . conAuthPurse) = (c . conAuthPurse))
         && (XiConPurse[cb, c, (cb . conAuthPurse)]))
        && ((c . ether) in (cb . ether)))
       && ((cb . archive) = (c . archive)))

Rule? (prop)
Applying propositional simplification,
this yields  3 subgoals:
Rbc_init.1.1 :

[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   ((cb . conAuthPurse) = (c . conAuthPurse))

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of Rbc_init.1.1.
Rbc_init.1.2 :

[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   (XiConPurse[cb, c, (cb . conAuthPurse)])

Rule? (hide-all-but (1 -4 -3))
Keeping (1 -4 -3) and hiding *,
this simplifies to:
Rbc_init.1.2 :

[-1]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-2]  (XiConPurse[c, cb, (c . conAuthPurse)])
  |-------
[1]   (XiConPurse[cb, c, (cb . conAuthPurse)])

Rule? (replace -1 :hide? t)
Replacing using formula -1,
this simplifies to:
Rbc_init.1.2 :

{-1}  (XiConPurse[c, cb, (cb . conAuthPurse)])
  |-------
[1]   (XiConPurse[cb, c, (cb . conAuthPurse)])

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of Rbc_init.1.2.
Rbc_init.1.3 :
```

```
[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   ((cb . archive) = (c . archive))

Rule? (assert)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of Rbc_init.1.3.
This completes the proof of Rbc_init.1.
Rbc_init.2 (TCC):

[-1]  (Concrete[c])
[-2]  (BetwInitState[cb])
[-3]  ((c . conAuthPurse) = (cb . conAuthPurse))
[-4]  (XiConPurse[c, cb, (c . conAuthPurse)])
[-5]  ((c . archive) = (cb . archive))
[-6]  ((c . ether) in (cb . ether))
  |-------
{1}   (cb in ConWorld)

Rule? (hide-all-but 1)
Keeping 1 and hiding *,
this simplifies to:
Rbc_init.2 :

  |-------
[1]   (cb in ConWorld)

Rule? (use "this?cbSubsetSig")
Using lemma this?cbSubsetSig,
this simplifies to:
Rbc_init.2 (TCC):

  |-------
{1}   (cb in cb)
[2]   (cb in ConWorld)

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of Rbc_init.2.

Q.E.D.
Run time  = 2.04 secs.
Real time = 41.89 secs.
```